Ceilometer Documentation

Release 23.0.0.0rc2.dev2

OpenStack Foundation

Sep 17, 2024

CONTENTS

The *Ceilometer* project is a data collection service that provides the ability to normalise and transform data across all current OpenStack core components with work underway to support future OpenStack components.

Ceilometer is a component of the Telemetry project. Its data can be used to provide customer billing, resource tracking, and alarming capabilities across all OpenStack core components.

This documentation offers information on how Ceilometer works and how to contribute to the project.

CHAPTER

ONE

OVERVIEW

1.1 Installation Guide

1.1.1 Telemetry Data Collection service overview

The Telemetry Data Collection services provide the following functions:

- Efficiently polls metering data related to OpenStack services.
- Collects event and metering data by monitoring notifications sent from services.
- Publishes collected data to various targets including data stores and message queues.

The Telemetry service consists of the following components:

A compute agent (`ceilometer-agent-compute`)
Runs on each compute node and polls for resource utilization statistics. This is actually the polling agent `ceilometer-polling` running with parameter `--polling-namespace compute`.

A central agent (`ceilometer-agent-central`)
Runs on a central management server to poll for resource utilization statistics for resources not tied to instances or compute nodes. Multiple agents can be started to scale service horizontally. This is actually the polling agent `ceilometer-polling` running with parameter `--polling-namespace central`.

A notification agent (`ceilometer-agent-notification`)
Runs on a central management server(s) and consumes messages from the message queue(s) to build event and metering data. Data is then published to defined targets. By default, data is pushed to Gnocchi.

These services communicate by using the OpenStack messaging bus. Ceilometer data is designed to be published to various endpoints for storage and analysis.

Note

Ceilometer previously provided a storage and API solution. As of Newton, this functionality is officially deprecated and discouraged. For efficient storage and statistical analysis of Ceilometer data, Gnocchi is recommended.

1.1.2 Install and Configure Controller Services

This section assumes that you already have a working OpenStack environment with at least the following components installed: Compute, Image Service, Identity.

Note that installation and configuration vary by distribution.

Ceilometer

Install and configure for openSUSE and SUSE Linux Enterprise

This section describes how to install and configure the Telemetry service, code-named ceilometer, on the controller node.

Prerequisites

Before you install and configure the Telemetry service, you must configure a target to send metering data to. The recommended endpoint is Gnocchi.

1. Source the `admin` credentials to gain access to admin-only CLI commands:

```
$ . admin-openrc
```

2. To create the service credentials, complete these steps:

   - Create the `ceilometer` user:

```
$ openstack user create --domain default --password-prompt ceilometer
User Password:
Repeat User Password:
+-----------+----------------------------------+
| Field     | Value                            |
+-----------+----------------------------------+
| domain_id | e0353a670a9e496da891347c589539e9 |
| enabled   | True                             |
| id        | c859c96f57bd4989a8ea1a0b1d8ff7cd |
| name      | ceilometer                       |
+-----------+----------------------------------+
```

   - Add the `admin` role to the `ceilometer` user.

```
$ openstack role add --project service --user ceilometer admin
```

Note

This command provides no output.

   - Create the `ceilometer` service entity:

```
$ openstack service create --name ceilometer \
  --description "Telemetry" metering
+-------------+----------------------------------+
| Field       | Value                            |
+-------------+----------------------------------+
| description | Telemetry                        |
| enabled     | True                             |
| id          | 5fb7fd1bb2954fddb378d4031c28c0e4 |
| name        | ceilometer                       |
| type        | metering                         |
+-------------+----------------------------------+
```

3. Register Gnocchi service in Keystone:

   - Create the `gnocchi` user:

```
$ openstack user create --domain default --password-prompt gnocchi
User Password:
Repeat User Password:
+-----------+----------------------------------+
| Field     | Value                            |
+-----------+----------------------------------+
| domain_id | e0353a670a9e496da891347c589539e9 |
| enabled   | True                             |
| id        | 8bacd064f6434ef2b6bbfbedb79b0318 |
| name      | gnocchi                          |
+-----------+----------------------------------+
```

   - Create the `gnocchi` service entity:

```
$ openstack service create --name gnocchi \
  --description "Metric Service" metric
+-------------+----------------------------------+
| Field       | Value                            |
+-------------+----------------------------------+
| description | Metric Service                   |
| enabled     | True                             |
| id          | 205978b411674e5a9990428f81d69384 |
| name        | gnocchi                          |
| type        | metric                           |
+-------------+----------------------------------+
```

   - Add the `admin` role to the `gnocchi` user.

```
$ openstack role add --project service --user gnocchi admin
```

Note

This command provides no output.

   - Create the Metric service API endpoints:

```
$ openstack endpoint create --region RegionOne \
  metric public http://controller:8041
+--------------+----------------------------------+
| Field        | Value                            |
+--------------+----------------------------------+
| enabled      | True                             |
| id           | b808b67b848d443e9eaaa5e5d796970c |
| interface    | public                           |
| region       | RegionOne                        |
| region_id    | RegionOne                        |
| service_id   | 205978b411674e5a9990428f81d69384 |
| service_name | gnocchi                          |
| service_type | metric                           |
| url          | http://controller:8041           |
+--------------+----------------------------------+

$ openstack endpoint create --region RegionOne \
  metric internal http://controller:8041
+--------------+----------------------------------+
| Field        | Value                            |
+--------------+----------------------------------+
| enabled      | True                             |
| id           | c7009b1c2ee54b71b771fa3d0ae4f948 |
| interface    | internal                         |
| region       | RegionOne                        |
| region_id    | RegionOne                        |
| service_id   | 205978b411674e5a9990428f81d69384 |
| service_name | gnocchi                          |
| service_type | metric                           |
| url          | http://controller:8041           |
+--------------+----------------------------------+

$ openstack endpoint create --region RegionOne \
  metric admin http://controller:8041
+--------------+----------------------------------+
| Field        | Value                            |
+--------------+----------------------------------+
| enabled      | True                             |
| id           | b2c00566d0604551b5fe1540c699db3d |
| interface    | admin                            |
| region       | RegionOne                        |
| region_id    | RegionOne                        |
| service_id   | 205978b411674e5a9990428f81d69384 |
| service_name | gnocchi                          |
| service_type | metric                           |
| url          | http://controller:8041           |
+--------------+----------------------------------+
```

Install Gnocchi

1. Install the Gnocchi packages. Alternatively, Gnocchi can be install using pip:

```
# zypper install openstack-gnocchi-api openstack-gnocchi-metricd \
  python-gnocchiclient
```

> **Note**
>
> Depending on your environment size, consider installing Gnocchi separately as it makes extensive use of the cpu.

2. Install the uWSGI packages. The following method uses operating system provided packages. Another alternative would be to use pip(or pip3, depending on the distribution); using pip is not described in this doc:

```
# zypper install uwsgi-plugin-python3 uwsgi
```

> **Note**
>
> Since the provided gnocchi-api wraps around uwsgi, you need to make sure that uWSGI is installed if you want to use gnocchi-api to run Gnocchi API. As Gnocchi API tier runs using WSGI, it can also alternatively be run using Apache httpd and mod_wsgi, or any other HTTP daemon.

3. Create the database for Gnocchi's indexer:

   - Use the database access client to connect to the database server as the `root` user:

   ```
   $ mysql -u root -p
   ```

   - Create the `gnocchi` database:

   ```
   CREATE DATABASE gnocchi;
   ```

   - Grant proper access to the `gnocchi` database:

   ```
   GRANT ALL PRIVILEGES ON gnocchi.* TO 'gnocchi'@'localhost' \
     IDENTIFIED BY 'GNOCCHI_DBPASS';
   GRANT ALL PRIVILEGES ON gnocchi.* TO 'gnocchi'@'%' \
     IDENTIFIED BY 'GNOCCHI_DBPASS';
   ```

   Replace `GNOCCHI_DBPASS` with a suitable password.

   - Exit the database access client.

4. Edit the `/etc/gnocchi/gnocchi.conf` file and add Keystone options:

   - In the `[api]` section, configure gnocchi to use keystone:

   ```
   [api]
   auth_mode = keystone
   ```

(continues on next page)

(continued from previous page)

```
port = 8041
uwsgi_mode = http-socket
```

- In the [keystone_authtoken] section, configure keystone authentication:

```
[keystone_authtoken]
...
auth_type = password
auth_url = http://controller:5000/v3
project_domain_name = Default
user_domain_name = Default
project_name = service
username = gnocchi
password = GNOCCHI_PASS
interface = internalURL
region_name = RegionOne
```

  Replace GNOCCHI_PASS with the password you chose for the gnocchi user in the Identity service.

- In the [indexer] section, configure database access:

```
[indexer]
url = mysql+pymysql://gnocchi:GNOCCHI_DBPASS@controller/gnocchi
```

  Replace GNOCCHI_DBPASS with the password you chose for Gnocchi's indexer database.

- In the [storage] section, configure location to store metric data. In this case, we will store it to the local file system. See Gnocchi documenation for a list of more durable and performant drivers:

```
[storage]
# coordination_url is not required but specifying one will improve
# performance with better workload division across workers.
coordination_url = redis://controller:6379
file_basepath = /var/lib/gnocchi
driver = file
```

5. Initialize Gnocchi:

```
gnocchi-upgrade
```

Finalize Gnocchi installation

1. Start the Gnocchi services and configure them to start when the system boots:

```
# systemctl enable openstack-gnocchi-api.service \
  openstack-gnocchi-metricd.service
# systemctl start openstack-gnocchi-api.service \
  openstack-gnocchi-metricd.service
```

Install and configure components

1. Install the packages:

```
# zypper install openstack-ceilometer-agent-notification \
  openstack-ceilometer-agent-central
```

2. Edit the `/etc/ceilometer/pipeline.yaml` file and complete the following section:

   - Configure Gnocchi connection:

```
publishers:
    # set address of Gnocchi
    # + filter out Gnocchi-related activity meters (Swift driver)
    # + set default archive policy
    - gnocchi://?filter_project=service&archive_policy=low
```

3. Edit the `/etc/ceilometer/ceilometer.conf` file and complete the following actions:

   - In the `[DEFAULT]` section, configure `RabbitMQ` message queue access:

```
[DEFAULT]
...
transport_url = rabbit://openstack:RABBIT_PASS@controller
```

   Replace `RABBIT_PASS` with the password you chose for the `openstack` account in `RabbitMQ`.

   - In the `[service_credentials]` section, configure service credentials:

```
[service_credentials]
...
auth_type = password
auth_url = http://controller:5000/v3
project_domain_id = default
user_domain_id = default
project_name = service
username = ceilometer
password = CEILOMETER_PASS
interface = internalURL
region_name = RegionOne
```

   Replace `CEILOMETER_PASS` with the password you chose for the `ceilometer` user in the Identity service.

4. Create Ceilometer resources in Gnocchi. Gnocchi should be running by this stage:

```
# ceilometer-upgrade
```

Finalize installation

1. Start the Telemetry services and configure them to start when the system boots:

```
# systemctl enable openstack-ceilometer-agent-notification.service \
  openstack-ceilometer-agent-central.service
# systemctl start openstack-ceilometer-agent-notification.service \
  openstack-ceilometer-agent-central.service
```

Install and configure for Red Hat Enterprise Linux and CentOS

This section describes how to install and configure the Telemetry service, code-named ceilometer, on the controller node.

Prerequisites

Before you install and configure the Telemetry service, you must configure a target to send metering data to. The recommended endpoint is Gnocchi.

1. Source the `admin` credentials to gain access to admin-only CLI commands:

```
$ . admin-openrc
```

2. To create the service credentials, complete these steps:

   - Create the `ceilometer` user:

```
$ openstack user create --domain default --password-prompt ceilometer
User Password:
Repeat User Password:
+-----------+----------------------------------+
| Field     | Value                            |
+-----------+----------------------------------+
| domain_id | e0353a670a9e496da891347c589539e9 |
| enabled   | True                             |
| id        | c859c96f57bd4989a8ea1a0b1d8ff7cd |
| name      | ceilometer                       |
+-----------+----------------------------------+
```

   - Add the `admin` role to the `ceilometer` user.

```
$ openstack role add --project service --user ceilometer admin
```

Note

This command provides no output.

- Create the `ceilometer` service entity:

```
$ openstack service create --name ceilometer \
  --description "Telemetry" metering
+-------------+----------------------------------+
| Field       | Value                            |
+-------------+----------------------------------+
| description | Telemetry                        |
| enabled     | True                             |
| id          | 5fb7fd1bb2954fddb378d4031c28c0e4 |
| name        | ceilometer                       |
| type        | metering                         |
+-------------+----------------------------------+
```

3. Register Gnocchi service in Keystone:

- Create the `gnocchi` user:

```
$ openstack user create --domain default --password-prompt gnocchi
User Password:
Repeat User Password:
+-----------+----------------------------------+
| Field     | Value                            |
+-----------+----------------------------------+
| domain_id | e0353a670a9e496da891347c589539e9 |
| enabled   | True                             |
| id        | 8bacd064f6434ef2b6bbfbedb79b0318 |
| name      | gnocchi                          |
+-----------+----------------------------------+
```

- Create the `gnocchi` service entity:

```
$ openstack service create --name gnocchi \
  --description "Metric Service" metric
+-------------+----------------------------------+
| Field       | Value                            |
+-------------+----------------------------------+
| description | Metric Service                   |
| enabled     | True                             |
| id          | 205978b411674e5a9990428f81d69384 |
| name        | gnocchi                          |
| type        | metric                           |
+-------------+----------------------------------+
```

- Add the `admin` role to the `gnocchi` user.

```
$ openstack role add --project service --user gnocchi admin
```

Note

This command provides no output.

- Create the Metric service API endpoints:

```
$ openstack endpoint create --region RegionOne \
  metric public http://controller:8041
+--------------+----------------------------------+
| Field        | Value                            |
+--------------+----------------------------------+
| enabled      | True                             |
| id           | b808b67b848d443e9eaaa5e5d796970c |
| interface    | public                           |
| region       | RegionOne                        |
| region_id    | RegionOne                        |
| service_id   | 205978b411674e5a9990428f81d69384 |
| service_name | gnocchi                          |
| service_type | metric                           |
| url          | http://controller:8041           |
+--------------+----------------------------------+

$ openstack endpoint create --region RegionOne \
  metric internal http://controller:8041
+--------------+----------------------------------+
| Field        | Value                            |
+--------------+----------------------------------+
| enabled      | True                             |
| id           | c7009b1c2ee54b71b771fa3d0ae4f948 |
| interface    | internal                         |
| region       | RegionOne                        |
| region_id    | RegionOne                        |
| service_id   | 205978b411674e5a9990428f81d69384 |
| service_name | gnocchi                          |
| service_type | metric                           |
| url          | http://controller:8041           |
+--------------+----------------------------------+

$ openstack endpoint create --region RegionOne \
  metric admin http://controller:8041
+--------------+----------------------------------+
| Field        | Value                            |
+--------------+----------------------------------+
| enabled      | True                             |
| id           | b2c00566d0604551b5fe1540c699db3d |
| interface    | admin                            |
| region       | RegionOne                        |
```

(continues on next page)

(continued from previous page)

```
| region_id    | RegionOne                        |
| service_id   | 205978b411674e5a9990428f81d69384 |
| service_name | gnocchi                          |
| service_type | metric                           |
| url          | http://controller:8041           |
+--------------+----------------------------------+
```

Install Gnocchi

1. Install the Gnocchi packages. Alternatively, Gnocchi can be install using pip:

```
# dnf install openstack-gnocchi-api openstack-gnocchi-metricd \
  python3-gnocchiclient
```

Note

Depending on your environment size, consider installing Gnocchi separately as it makes extensive use of the cpu.

2. Install the uWSGI packages. The following method uses operating system provided packages. Another alternative would be to use pip(or pip3, depending on the distribution); using pip is not described in this doc:

```
# dnf install uwsgi-plugin-common uwsgi-plugin-python3 uwsgi
```

Note

Since the provided gnocchi-api wraps around uwsgi, you need to make sure that uWSGI is installed if you want to use gnocchi-api to run Gnocchi API. As Gnocchi API tier runs using WSGI, it can also alternatively be run using Apache httpd and mod_wsgi, or any other HTTP daemon.

3. Create the database for Gnocchi's indexer:

   - Use the database access client to connect to the database server as the `root` user:

```
$ mysql -u root -p
```

   - Create the `gnocchi` database:

```
CREATE DATABASE gnocchi;
```

   - Grant proper access to the `gnocchi` database:

```
GRANT ALL PRIVILEGES ON gnocchi.* TO 'gnocchi'@'localhost' \
  IDENTIFIED BY 'GNOCCHI_DBPASS';
GRANT ALL PRIVILEGES ON gnocchi.* TO 'gnocchi'@'%' \
  IDENTIFIED BY 'GNOCCHI_DBPASS';
```

Replace GNOCCHI_DBPASS with a suitable password.

- Exit the database access client.

4. Edit the /etc/gnocchi/gnocchi.conf file and add Keystone options:

   - In the [api] section, configure gnocchi to use keystone:

```
[api]
auth_mode = keystone
port = 8041
uwsgi_mode = http-socket
```

   - In the [keystone_authtoken] section, configure keystone authentication:

```
[keystone_authtoken]
...
auth_type = password
auth_url = http://controller:5000/v3
project_domain_name = Default
user_domain_name = Default
project_name = service
username = gnocchi
password = GNOCCHI_PASS
interface = internalURL
region_name = RegionOne
```

   Replace GNOCCHI_PASS with the password you chose for the gnocchi user in the Identity service.

   - In the [indexer] section, configure database access:

```
[indexer]
url = mysql+pymysql://gnocchi:GNOCCHI_DBPASS@controller/gnocchi
```

   Replace GNOCCHI_DBPASS with the password you chose for Gnocchi's indexer database.

   - In the [storage] section, configure location to store metric data. In this case, we will store it to the local file system. See Gnocchi documenation for a list of more durable and performant drivers:

```
[storage]
# coordination_url is not required but specifying one will improve
# performance with better workload division across workers.
coordination_url = redis://controller:6379
file_basepath = /var/lib/gnocchi
driver = file
```

5. Initialize Gnocchi:

```
gnocchi-upgrade
```

Finalize Gnocchi installation

1. Start the Gnocchi services and configure them to start when the system boots:

```
# systemctl enable openstack-gnocchi-api.service \
  openstack-gnocchi-metricd.service
# systemctl start openstack-gnocchi-api.service \
  openstack-gnocchi-metricd.service
```

Install and configure components

1. Install the Ceilometer packages:

```
# dnf install openstack-ceilometer-notification \
  openstack-ceilometer-central
```

2. Edit the `/etc/ceilometer/pipeline.yaml` file and complete the following section:

   - Configure Gnocchi connection:

```
publishers:
    # set address of Gnocchi
    # + filter out Gnocchi-related activity meters (Swift driver)
    # + set default archive policy
    - gnocchi://?filter_project=service&archive_policy=low
```

3. Edit the `/etc/ceilometer/ceilometer.conf` file and complete the following actions:

   - In the `[DEFAULT]` section, configure `RabbitMQ` message queue access:

```
[DEFAULT]
...
transport_url = rabbit://openstack:RABBIT_PASS@controller
```

   Replace `RABBIT_PASS` with the password you chose for the `openstack` account in `RabbitMQ`.

   - In the `[service_credentials]` section, configure service credentials:

```
[service_credentials]
...
auth_type = password
auth_url = http://controller:5000/v3
project_domain_id = default
user_domain_id = default
project_name = service
username = ceilometer
password = CEILOMETER_PASS
interface = internalURL
region_name = RegionOne
```

   Replace `CEILOMETER_PASS` with the password you chose for the `ceilometer` user in the Identity service.

4. Create Ceilometer resources in Gnocchi. Gnocchi should be running by this stage:

```
# ceilometer-upgrade
```

Finalize installation

1. Start the Telemetry services and configure them to start when the system boots:

```
# systemctl enable openstack-ceilometer-notification.service \
  openstack-ceilometer-central.service
# systemctl start openstack-ceilometer-notification.service \
  openstack-ceilometer-central.service
```

Install and configure for Ubuntu

This section describes how to install and configure the Telemetry service, code-named ceilometer, on the controller node.

Prerequisites

Before you install and configure the Telemetry service, you must configure a target to send metering data to. The recommended endpoint is Gnocchi.

1. Source the `admin` credentials to gain access to admin-only CLI commands:

```
$ . admin-openrc
```

2. To create the service credentials, complete these steps:

   - Create the `ceilometer` user:

```
$ openstack user create --domain default --password-prompt ceilometer
User Password:
Repeat User Password:
+-----------+----------------------------------+
| Field     | Value                            |
+-----------+----------------------------------+
| domain_id | e0353a670a9e496da891347c589539e9 |
| enabled   | True                             |
| id        | c859c96f57bd4989a8ea1a0b1d8ff7cd |
| name      | ceilometer                       |
+-----------+----------------------------------+
```

   - Add the `admin` role to the `ceilometer` user.

```
$ openstack role add --project service --user ceilometer admin
```

Note

This command provides no output.

- Create the `ceilometer` service entity:

```
$ openstack service create --name ceilometer \
  --description "Telemetry" metering
+-------------+----------------------------------+
| Field       | Value                            |
+-------------+----------------------------------+
| description | Telemetry                        |
| enabled     | True                             |
| id          | 5fb7fd1bb2954fddb378d4031c28c0e4 |
| name        | ceilometer                       |
| type        | metering                         |
+-------------+----------------------------------+
```

3. Register Gnocchi service in Keystone:

- Create the `gnocchi` user:

```
$ openstack user create --domain default --password-prompt gnocchi
User Password:
Repeat User Password:
+-----------+----------------------------------+
| Field     | Value                            |
+-----------+----------------------------------+
| domain_id | e0353a670a9e496da891347c589539e9 |
| enabled   | True                             |
| id        | 8bacd064f6434ef2b6bbfbedb79b0318 |
| name      | gnocchi                          |
+-----------+----------------------------------+
```

- Create the `gnocchi` service entity:

```
$ openstack service create --name gnocchi \
  --description "Metric Service" metric
+-------------+----------------------------------+
| Field       | Value                            |
+-------------+----------------------------------+
| description | Metric Service                   |
| enabled     | True                             |
| id          | 205978b411674e5a9990428f81d69384 |
| name        | gnocchi                          |
| type        | metric                           |
+-------------+----------------------------------+
```

- Add the `admin` role to the `gnocchi` user.

```
$ openstack role add --project service --user gnocchi admin
```

Note

This command provides no output.

- Create the Metric service API endpoints:

```
$ openstack endpoint create --region RegionOne \
  metric public http://controller:8041
+--------------+----------------------------------+
| Field        | Value                            |
+--------------+----------------------------------+
| enabled      | True                             |
| id           | b808b67b848d443e9eaaa5e5d796970c |
| interface    | public                           |
| region       | RegionOne                        |
| region_id    | RegionOne                        |
| service_id   | 205978b411674e5a9990428f81d69384 |
| service_name | gnocchi                          |
| service_type | metric                           |
| url          | http://controller:8041           |
+--------------+----------------------------------+

$ openstack endpoint create --region RegionOne \
  metric internal http://controller:8041
+--------------+----------------------------------+
| Field        | Value                            |
+--------------+----------------------------------+
| enabled      | True                             |
| id           | c7009b1c2ee54b71b771fa3d0ae4f948 |
| interface    | internal                         |
| region       | RegionOne                        |
| region_id    | RegionOne                        |
| service_id   | 205978b411674e5a9990428f81d69384 |
| service_name | gnocchi                          |
| service_type | metric                           |
| url          | http://controller:8041           |
+--------------+----------------------------------+

$ openstack endpoint create --region RegionOne \
  metric admin http://controller:8041
+--------------+----------------------------------+
| Field        | Value                            |
+--------------+----------------------------------+
| enabled      | True                             |
| id           | b2c00566d0604551b5fe1540c699db3d |
| interface    | admin                            |
| region       | RegionOne                        |
```

(continues on next page)

(continued from previous page)

```
| region_id    | RegionOne                        |
| service_id   | 205978b411674e5a9990428f81d69384 |
| service_name | gnocchi                          |
| service_type | metric                           |
| url          | http://controller:8041           |
+--------------+----------------------------------+
```

Install Gnocchi

1. Install the Gnocchi packages. Alternatively, Gnocchi can be installed using pip:

```
# apt-get install gnocchi-api gnocchi-metricd python-gnocchiclient
```

Note

Depending on your environment size, consider installing Gnocchi separately as it makes extensive use of the cpu.

2. Install the uWSGI packages. The following method uses operating system provided packages. Another alternative would be to use pip(or pip3, depending on the distribution); using pip is not described in this doc:

```
# apt-get install uwsgi-plugin-python3 uwsgi
```

Note

Since the provided gnocchi-api wraps around uwsgi, you need to make sure that uWSGI is installed if you want to use gnocchi-api to run Gnocchi API. As Gnocchi API tier runs using WSGI, it can also alternatively be run using Apache httpd and mod_wsgi, or any other HTTP daemon.

3. Create the database for Gnocchi's indexer:

   - Use the database access client to connect to the database server as the `root` user:

```
$ mysql -u root -p
```

   - Create the `gnocchi` database:

```
CREATE DATABASE gnocchi;
```

   - Grant proper access to the `gnocchi` database:

```
GRANT ALL PRIVILEGES ON gnocchi.* TO 'gnocchi'@'localhost' \
  IDENTIFIED BY 'GNOCCHI_DBPASS';
GRANT ALL PRIVILEGES ON gnocchi.* TO 'gnocchi'@'%' \
  IDENTIFIED BY 'GNOCCHI_DBPASS';
```

   Replace `GNOCCHI_DBPASS` with a suitable password.

- Exit the database access client.

4. Edit the /etc/gnocchi/gnocchi.conf file and add Keystone options:

   - In the [api] section, configure gnocchi to use keystone:

```
[api]
auth_mode = keystone
port = 8041
uwsgi_mode = http-socket
```

   - In the [keystone_authtoken] section, configure keystone authentication:

```
[keystone_authtoken]
...
auth_type = password
auth_url = http://controller:5000/v3
project_domain_name = Default
user_domain_name = Default
project_name = service
username = gnocchi
password = GNOCCHI_PASS
interface = internalURL
region_name = RegionOne
```

   Replace GNOCCHI_PASS with the password you chose for the gnocchi user in the Identity service.

   - In the [indexer] section, configure database access:

```
[indexer]
url = mysql+pymysql://gnocchi:GNOCCHI_DBPASS@controller/gnocchi
```

   Replace GNOCCHI_DBPASS with the password you chose for Gnocchi's indexer database.

   - In the [storage] section, configure location to store metric data. In this case, we will store it to the local file system. See Gnocchi documenation for a list of more durable and performant drivers:

```
[storage]
# coordination_url is not required but specifying one will improve
# performance with better workload division across workers.
coordination_url = redis://controller:6379
file_basepath = /var/lib/gnocchi
driver = file
```

5. Initialize Gnocchi:

```
gnocchi-upgrade
```

Finalize Gnocchi installation

1. Restart the Gnocchi services:

```
# service gnocchi-api restart
# service gnocchi-metricd restart
```

Install and configure components

1. Install the ceilometer packages:

```
# apt-get install ceilometer-agent-notification \
  ceilometer-agent-central
```

2. Edit the `/etc/ceilometer/pipeline.yaml` file and complete the following section:

   - Configure Gnocchi connection:

```
publishers:
    # set address of Gnocchi
    # + filter out Gnocchi-related activity meters (Swift driver)
    # + set default archive policy
    - gnocchi://?filter_project=service&archive_policy=low
```

3. Edit the `/etc/ceilometer/ceilometer.conf` file and complete the following actions:

   - In the `[DEFAULT]` section, configure `RabbitMQ` message queue access:

```
[DEFAULT]
...
transport_url = rabbit://openstack:RABBIT_PASS@controller
```

   Replace `RABBIT_PASS` with the password you chose for the `openstack` account in `RabbitMQ`.

   - In the `[service_credentials]` section, configure service credentials:

```
[service_credentials]
...
auth_type = password
auth_url = http://controller:5000/v3
project_domain_id = default
user_domain_id = default
project_name = service
username = ceilometer
password = CEILOMETER_PASS
interface = internalURL
region_name = RegionOne
```

   Replace `CEILOMETER_PASS` with the password you chose for the `ceilometer` user in the Identity service.

4. Create Ceilometer resources in Gnocchi. Gnocchi should be running by this stage:

```
# ceilometer-upgrade
```

Finalize installation

1. Restart the Telemetry services:

```
# service ceilometer-agent-central restart
# service ceilometer-agent-notification restart
```

Additional steps are required to configure services to interact with ceilometer:

Cinder

Enable Block Storage meters for openSUSE and SUSE Linux Enterprise

Telemetry uses notifications to collect Block Storage service meters. Perform these steps on the controller and Block Storage nodes.

> **Note**
>
> Your environment must include the Block Storage service.

Configure Cinder to use Telemetry

Edit the `/etc/cinder/cinder.conf` file and complete the following actions:

- In the `[oslo_messaging_notifications]` section, configure notifications:

```
[oslo_messaging_notifications]
...
driver = messagingv2
```

- Enable periodic usage statistics relating to block storage. To use it, you must run this command in the following format:

```
$ cinder-volume-usage-audit  --start_time='YYYY-MM-DD HH:MM:SS' \
  --end_time='YYYY-MM-DD HH:MM:SS' --send_actions
```

  This script outputs what volumes or snapshots were created, deleted, or exists in a given period of time and some information about these volumes or snapshots.

  Using this script via cron you can get notifications periodically, for example, every 5 minutes:

```
*/5 * * * * /path/to/cinder-volume-usage-audit --send_actions
```

Finalize installation

1. Restart the Block Storage services on the controller node:

```
# systemctl restart openstack-cinder-api.service openstack-cinder-
↪scheduler.service
```

2. Restart the Block Storage services on the storage nodes:

```
# systemctl restart openstack-cinder-volume.service
```

Enable Block Storage meters for Red Hat Enterprise Linux and CentOS

Telemetry uses notifications to collect Block Storage service meters. Perform these steps on the controller and Block Storage nodes.

Note

Your environment must include the Block Storage service.

Configure Cinder to use Telemetry

Edit the `/etc/cinder/cinder.conf` file and complete the following actions:

- In the `[oslo_messaging_notifications]` section, configure notifications:

```
[oslo_messaging_notifications]
...
driver = messagingv2
```

- Enable periodic usage statistics relating to block storage. To use it, you must run this command in the following format:

```
$ cinder-volume-usage-audit  --start_time='YYYY-MM-DD HH:MM:SS' \
  --end_time='YYYY-MM-DD HH:MM:SS' --send_actions
```

  This script outputs what volumes or snapshots were created, deleted, or exists in a given period of time and some information about these volumes or snapshots.

  Using this script via cron you can get notifications periodically, for example, every 5 minutes:

```
*/5 * * * * /path/to/cinder-volume-usage-audit --send_actions
```

Finalize installation

1. Restart the Block Storage services on the controller node:

```
# systemctl restart openstack-cinder-api.service openstack-cinder-
↪scheduler.service
```

2. Restart the Block Storage services on the storage nodes:

```
# systemctl restart openstack-cinder-volume.service
```

Enable Block Storage meters for Ubuntu

Telemetry uses notifications to collect Block Storage service meters. Perform these steps on the controller and Block Storage nodes.

Note

Your environment must include the Block Storage service.

Configure Cinder to use Telemetry

Edit the `/etc/cinder/cinder.conf` file and complete the following actions:

- In the `[oslo_messaging_notifications]` section, configure notifications:

```
[oslo_messaging_notifications]
...
driver = messagingv2
```

- Enable periodic usage statistics relating to block storage. To use it, you must run this command in the following format:

```
$ cinder-volume-usage-audit  --start_time='YYYY-MM-DD HH:MM:SS' \
  --end_time='YYYY-MM-DD HH:MM:SS' --send_actions
```

  This script outputs what volumes or snapshots were created, deleted, or exists in a given period of time and some information about these volumes or snapshots.

  Using this script via cron you can get notifications periodically, for example, every 5 minutes:

```
*/5 * * * * /path/to/cinder-volume-usage-audit --send_actions
```

Finalize installation

1. Restart the Block Storage services on the controller node:

```
# service cinder-api restart
# service cinder-scheduler restart
```

2. Restart the Block Storage services on the storage nodes:

```
# service cinder-volume restart
```

Glance

Enable Image service meters for openSUSE and SUSE Linux Enterprise

Telemetry uses notifications to collect Image service meters. Perform these steps on the controller node.

Configure the Image service to use Telemetry

- Edit the `/etc/glance/glance-api.conf` file and complete the following actions:
  - In the `[DEFAULT]`, `[oslo_messaging_notifications]` sections, configure notifications and RabbitMQ message broker access:

    ```
    [DEFAULT]
    ...
    transport_url = rabbit://openstack:RABBIT_PASS@controller

    [oslo_messaging_notifications]
    ...
    driver = messagingv2
    ```

    Replace `RABBIT_PASS` with the password you chose for the `openstack` account in `RabbitMQ`.

Finalize installation

- Restart the Image service:

```
# systemctl restart openstack-glance-api.service
```

Enable Image service meters for Red Hat Enterprise Linux and CentOS

Telemetry uses notifications to collect Image service meters. Perform these steps on the controller node.

Configure the Image service to use Telemetry

- Edit the `/etc/glance/glance-api.conf` file and complete the following actions:
  - In the `[DEFAULT]`, `[oslo_messaging_notifications]` sections, configure notifications and RabbitMQ message broker access:

    ```
    [DEFAULT]
    ...
    transport_url = rabbit://openstack:RABBIT_PASS@controller

    [oslo_messaging_notifications]
    ...
    driver = messagingv2
    ```

    Replace `RABBIT_PASS` with the password you chose for the `openstack` account in `RabbitMQ`.

Finalize installation

- Restart the Image service:

```
# systemctl restart openstack-glance-api.service
```

Enable Image service meters for Ubuntu

Telemetry uses notifications to collect Image service meters. Perform these steps on the controller node.

Configure the Image service to use Telemetry

- Edit the `/etc/glance/glance-api.conf` file and complete the following actions:
  - In the `[DEFAULT]`, `[oslo_messaging_notifications]` sections, configure notifications and RabbitMQ message broker access:

    ```
    [DEFAULT]
    ...
    transport_url = rabbit://openstack:RABBIT_PASS@controller

    [oslo_messaging_notifications]
    ...
    driver = messagingv2
    ```

    Replace `RABBIT_PASS` with the password you chose for the `openstack` account in `RabbitMQ`.

Finalize installation

- Restart the Image service:

```
# service glance-api restart
```

Heat

Enable Orchestration service meters for openSUSE and SUSE Linux Enterprise

Telemetry uses notifications to collect Orchestration service meters. Perform these steps on the controller node.

Configure the Orchestration service to use Telemetry

- Edit the `/etc/heat/heat.conf` and complete the following actions:
  - In the `[oslo_messaging_notifications]` sections, enable notifications:

    ```
    [oslo_messaging_notifications]
    ...
    driver = messagingv2
    ```

Finalize installation

- Restart the Orchestration service:

```
# systemctl restart openstack-heat-api.service \
  openstack-heat-api-cfn.service openstack-heat-engine.service
```

Enable Orchestration service meters for Red Hat Enterprise Linux and CentOS

Telemetry uses notifications to collect Orchestration service meters. Perform these steps on the controller node.

Configure the Orchestration service to use Telemetry

- Edit the `/etc/heat/heat.conf` and complete the following actions:
  - In the `[oslo_messaging_notifications]` sections, enable notifications:

    ```
    [oslo_messaging_notifications]
    ...
    driver = messagingv2
    ```

Finalize installation

- Restart the Orchestration service:

```
# systemctl restart openstack-heat-api.service \
  openstack-heat-api-cfn.service openstack-heat-engine.service
```

Enable Orchestration service meters for Ubuntu

Telemetry uses notifications to collect Orchestration service meters. Perform these steps on the controller node.

Configure the Orchestration service to use Telemetry

- Edit the `/etc/heat/heat.conf` and complete the following actions:
  - In the `[oslo_messaging_notifications]` sections, enable notifications:

```
[oslo_messaging_notifications]
...
driver = messagingv2
```

Finalize installation

- Restart the Orchestration service:

```
# service heat-api restart
# service heat-api-cfn restart
# service heat-engine restart
```

Keystone

To enable auditing of API requests, Keystone provides middleware which captures API requests to a service and emits data to Ceilometer. Instructions to enable this functionality is available in Keystone's developer documentation. Ceilometer will captures this information as `audit.http.*` events.

Neutron

Enable Networking service meters for openSUSE and SUSE Linux Enterprise

Telemetry uses notifications to collect Networking service meters. Perform these steps on the controller node.

Configure the Networking service to use Telemetry

- Edit the /etc/neutron/neutron.conf and complete the following actions:
  - In the [oslo_messaging_notifications] sections, enable notifications:

    ```
    [oslo_messaging_notifications]
    ...
    driver = messagingv2
    ```

Finalize installation

- Restart the Networking service:

```
# systemctl restart neutron-server.service
```

Enable Networking service meters for Red Hat Enterprise Linux and CentOS

Telemetry uses notifications to collect Networking service meters. Perform these steps on the controller node.

Configure the Networking service to use Telemetry

- Edit the /etc/neutron/neutron.conf and complete the following actions:
  - In the [oslo_messaging_notifications] sections, enable notifications:

    ```
    [oslo_messaging_notifications]
    ...
    driver = messagingv2
    ```

Finalize installation

- Restart the Networking service:

```
# systemctl restart neutron-server.service
```

Enable Networking service meters for Ubuntu

Telemetry uses notifications to collect Networking service meters. Perform these steps on the controller node.

Configure the Networking service to use Telemetry

- Edit the `/etc/neutron/neutron.conf` and complete the following actions:
  - In the `[oslo_messaging_notifications]` sections, enable notifications:

    ```
    [oslo_messaging_notifications]
    ...
    driver = messagingv2
    ```

Finalize installation

- Restart the Networking service:

```
# service neutron-server restart
```

Swift

Enable Object Storage meters for openSUSE and SUSE Linux Enterprise

Telemetry uses a combination of polling and notifications to collect Object Storage meters.

Note

Your environment must include the Object Storage service.

Prerequisites

The Telemetry service requires access to the Object Storage service using the `ResellerAdmin` role. Perform these steps on the controller node.

1. Source the `admin` credentials to gain access to admin-only CLI commands.

```
$ . admin-openrc
```

2. Create the `ResellerAdmin` role:

```
$ openstack role create ResellerAdmin
+-----------+----------------------------------+
| Field     | Value                            |
+-----------+----------------------------------+
| domain_id | None                             |
| id        | 462fa46c13fd4798a95a3bfbe27b5e54 |
| name      | ResellerAdmin                    |
+-----------+----------------------------------+
```

3. Add the `ResellerAdmin` role to the `ceilometer` user:

```
$ openstack role add --project service --user ceilometer ResellerAdmin
```

Note

This command provides no output.

Install components

- Install the packages:

```
# zypper install python-ceilometermiddleware
```

Configure Object Storage to use Telemetry

Perform these steps on the controller and any other nodes that run the Object Storage proxy service.

- Edit the `/etc/swift/proxy-server.conf` file and complete the following actions:
  - In the `[filter:keystoneauth]` section, add the `ResellerAdmin` role:

```
[filter:keystoneauth]
...
operator_roles = admin, user, ResellerAdmin
```

  - In the `[pipeline:main]` section, add `ceilometer`:

```
[pipeline:main]
pipeline = catch_errors gatekeeper healthcheck proxy-logging cache␣
↪container_sync bulk ratelimit authtoken keystoneauth container-
↪quotas account-quotas slo dlo versioned_writes proxy-logging␣
↪ceilometer proxy-server
```

  - In the `[filter:ceilometer]` section, configure notifications:

```
[filter:ceilometer]
paste.filter_factory = ceilometermiddleware.swift:filter_factory
...
control_exchange = swift
url = rabbit://openstack:RABBIT_PASS@controller:5672/
driver = messagingv2
topic = notifications
log_level = WARN
```

    Replace `RABBIT_PASS` with the password you chose for the `openstack` account in `RabbitMQ`.

Finalize installation

- Restart the Object Storage proxy service:

```
# systemctl restart openstack-swift-proxy.service
```

Enable Object Storage meters for Red Hat Enterprise Linux and CentOS

Telemetry uses a combination of polling and notifications to collect Object Storage meters.

Note

Your environment must include the Object Storage service.

Prerequisites

The Telemetry service requires access to the Object Storage service using the `ResellerAdmin` role. Perform these steps on the controller node.

1. Source the `admin` credentials to gain access to admin-only CLI commands.

```
$ . admin-openrc
```

2. Create the `ResellerAdmin` role:

```
$ openstack role create ResellerAdmin
+-----------+----------------------------------+
| Field     | Value                            |
+-----------+----------------------------------+
| domain_id | None                             |
| id        | 462fa46c13fd4798a95a3bfbe27b5e54 |
| name      | ResellerAdmin                    |
+-----------+----------------------------------+
```

3. Add the `ResellerAdmin` role to the `ceilometer` user:

```
$ openstack role add --project service --user ceilometer ResellerAdmin
```

Note

This command provides no output.

Install components

- Install the packages:

```
# dnf install python3-ceilometermiddleware
```

Configure Object Storage to use Telemetry

Perform these steps on the controller and any other nodes that run the Object Storage proxy service.

- Edit the /etc/swift/proxy-server.conf file and complete the following actions:
  - In the [filter:keystoneauth] section, add the ResellerAdmin role:

```
[filter:keystoneauth]
...
operator_roles = admin, user, ResellerAdmin
```

  - In the [pipeline:main] section, add ceilometer:

```
[pipeline:main]
pipeline = catch_errors gatekeeper healthcheck proxy-logging cache␣
↪container_sync bulk ratelimit authtoken keystoneauth container-
↪quotas account-quotas slo dlo versioned_writes proxy-logging␣
↪ceilometer proxy-server
```

  - In the [filter:ceilometer] section, configure notifications:

```
[filter:ceilometer]
paste.filter_factory = ceilometermiddleware.swift:filter_factory
...
control_exchange = swift
url = rabbit://openstack:RABBIT_PASS@controller:5672/
driver = messagingv2
topic = notifications
log_level = WARN
```

    Replace RABBIT_PASS with the password you chose for the openstack account in RabbitMQ.

Finalize installation

- Restart the Object Storage proxy service:

```
# systemctl restart openstack-swift-proxy.service
```

Enable Object Storage meters for Ubuntu

Telemetry uses a combination of polling and notifications to collect Object Storage meters.

Note

Your environment must include the Object Storage service.

Prerequisites

The Telemetry service requires access to the Object Storage service using the `ResellerAdmin` role. Perform these steps on the controller node.

1. Source the `admin` credentials to gain access to admin-only CLI commands.

```
$ . admin-openrc
```

2. Create the `ResellerAdmin` role:

```
$ openstack role create ResellerAdmin
+-----------+----------------------------------+
| Field     | Value                            |
+-----------+----------------------------------+
| domain_id | None                             |
| id        | 462fa46c13fd4798a95a3bfbe27b5e54 |
| name      | ResellerAdmin                    |
+-----------+----------------------------------+
```

3. Add the `ResellerAdmin` role to the `ceilometer` user:

```
$ openstack role add --project service --user ceilometer ResellerAdmin
```

Note

This command provides no output.

Install components

- Install the packages:

```
# apt-get install python-ceilometermiddleware
```

Configure Object Storage to use Telemetry

Perform these steps on the controller and any other nodes that run the Object Storage proxy service.

- Edit the `/etc/swift/proxy-server.conf` file and complete the following actions:
  - In the `[filter:keystoneauth]` section, add the `ResellerAdmin` role:

    ```
    [filter:keystoneauth]
    ...
    operator_roles = admin, user, ResellerAdmin
    ```

  - In the `[pipeline:main]` section, add `ceilometer`:

    ```
    [pipeline:main]
    pipeline = catch_errors gatekeeper healthcheck proxy-logging cache␣
    ↪container_sync bulk ratelimit authtoken keystoneauth container-
    ↪quotas account-quotas slo dlo versioned_writes proxy-logging␣
    ↪ceilometer proxy-server
    ```

  - In the `[filter:ceilometer]` section, configure notifications:

    ```
    [filter:ceilometer]
    paste.filter_factory = ceilometermiddleware.swift:filter_factory
    ...
    control_exchange = swift
    url = rabbit://openstack:RABBIT_PASS@controller:5672/
    driver = messagingv2
    topic = notifications
    log_level = WARN
    ```

    Replace `RABBIT_PASS` with the password you chose for the `openstack` account in `RabbitMQ`.

Finalize installation

- Restart the Object Storage proxy service:

  ```
  # service swift-proxy restart
  ```

1.1.3 Install and Configure Compute Services

This section assumes that you already have a working OpenStack environment with at least the following components installed: Compute, Image Service, Identity.

Note that installation and configuration vary by distribution.

Enable Compute service meters for openSUSE and SUSE Linux Enterprise

Telemetry uses a combination of notifications and an agent to collect Compute meters. Perform these steps on each compute node.

Install and configure components

1. Install the packages:

```
# zypper install openstack-ceilometer-agent-compute
# zypper install openstack-ceilometer-agent-ipmi (optional)
```

2. Edit the `/etc/ceilometer/ceilometer.conf` file and complete the following actions:

   - In the `[DEFAULT]` section, configure `RabbitMQ` message queue access:

   ```
   [DEFAULT]
   ...
   transport_url = rabbit://openstack:RABBIT_PASS@controller
   ```

   Replace `RABBIT_PASS` with the password you chose for the `openstack` account in `RabbitMQ`.

   - In the `[service_credentials]` section, configure service credentials:

   ```
   [service_credentials]
   ...
   auth_url = http://controller:5000
   project_domain_id = default
   user_domain_id = default
   auth_type = password
   username = ceilometer
   project_name = service
   password = CEILOMETER_PASS
   interface = internalURL
   region_name = RegionOne
   ```

   Replace `CEILOMETER_PASS` with the password you chose for the `ceilometer` user in the Identity service.

Configure Compute to use Telemetry

- Edit the `/etc/nova/nova.conf` file and configure notifications in the `[DEFAULT]` section:

```
[DEFAULT]
...
instance_usage_audit = True
instance_usage_audit_period = hour

[notifications]
...
```

(continues on next page)

(continued from previous page)

```
notify_on_state_change = vm_and_task_state

[oslo_messaging_notifications]
...
driver = messagingv2
```

Configure Compute to poll IPMI meters

Note

To enable IPMI meters, ensure IPMITool is installed and the host supports Intel Node Manager.

- Edit the `/etc/sudoers` file and include:

```
ceilometer ALL = (root) NOPASSWD: /usr/bin/ceilometer-rootwrap /etc/
↪ceilometer/rootwrap.conf *
```

- Edit the `/etc/ceilometer/polling.yaml` to include the required meters, for example:

```
- name: ipmi
  interval: 300
  meters:
    - hardware.ipmi.temperature
```

Finalize installation

1. Start the agent and configure it to start when the system boots:

```
# systemctl enable openstack-ceilometer-agent-compute.service
# systemctl start openstack-ceilometer-agent-compute.service
# systemctl enable openstack-ceilometer-agent-ipmi.service (optional)
# systemctl start openstack-ceilometer-agent-ipmi.service (optional)
```

2. Restart the Compute service:

```
# systemctl restart openstack-nova-compute.service
```

Enable Compute service meters for Red Hat Enterprise Linux and CentOS

Telemetry uses a combination of notifications and an agent to collect Compute meters. Perform these steps on each compute node.

Install and configure components

1. Install the packages:

```
# dnf install openstack-ceilometer-compute
# dnf install openstack-ceilometer-ipmi (optional)
```

2. Edit the `/etc/ceilometer/ceilometer.conf` file and complete the following actions:

   - In the `[DEFAULT]` section, configure `RabbitMQ` message queue access:

   ```
   [DEFAULT]
   ...
   transport_url = rabbit://openstack:RABBIT_PASS@controller
   ```

   Replace `RABBIT_PASS` with the password you chose for the `openstack` account in `RabbitMQ`.

   - In the `[service_credentials]` section, configure service credentials:

   ```
   [service_credentials]
   ...
   auth_url = http://controller:5000
   project_domain_id = default
   user_domain_id = default
   auth_type = password
   username = ceilometer
   project_name = service
   password = CEILOMETER_PASS
   interface = internalURL
   region_name = RegionOne
   ```

   Replace `CEILOMETER_PASS` with the password you chose for the `ceilometer` user in the Identity service.

Configure Compute to use Telemetry

- Edit the `/etc/nova/nova.conf` file and configure notifications in the `[DEFAULT]` section:

```
[DEFAULT]
...
instance_usage_audit = True
instance_usage_audit_period = hour

[notifications]
...
```

(continues on next page)

(continued from previous page)

```
notify_on_state_change = vm_and_task_state

[oslo_messaging_notifications]
...
driver = messagingv2
```

Configure Compute to poll IPMI meters

Note

To enable IPMI meters, ensure IPMITool is installed and the host supports Intel Node Manager.

- Edit the `/etc/sudoers` file and include:

```
ceilometer ALL = (root) NOPASSWD: /usr/bin/ceilometer-rootwrap /etc/
↪ceilometer/rootwrap.conf *
```

- Edit the `/etc/ceilometer/polling.yaml` to include the required meters, for example:

```
- name: ipmi
  interval: 300
  meters:
    - hardware.ipmi.temperature
```

Finalize installation

1. Start the agent and configure it to start when the system boots:

```
# systemctl enable openstack-ceilometer-compute.service
# systemctl start openstack-ceilometer-compute.service
# systemctl enable openstack-ceilometer-ipmi.service (optional)
# systemctl start openstack-ceilometer-ipmi.service (optional)
```

2. Restart the Compute service:

```
# systemctl restart openstack-nova-compute.service
```

Enable Compute service meters for Ubuntu

Telemetry uses a combination of notifications and an agent to collect Compute meters. Perform these steps on each compute node.

Install and configure components

1. Install the packages:

```
# apt-get install ceilometer-agent-compute
# apt-get install ceilometer-agent-ipmi (optional)
```

2. Edit the /etc/ceilometer/ceilometer.conf file and complete the following actions:

   - In the [DEFAULT] section, configure RabbitMQ message queue access:

   ```
   [DEFAULT]
   ...
   transport_url = rabbit://openstack:RABBIT_PASS@controller
   ```

   Replace RABBIT_PASS with the password you chose for the openstack account in RabbitMQ.

   - In the [service_credentials] section, configure service credentials:

   ```
   [service_credentials]
   ...
   auth_url = http://controller:5000
   project_domain_id = default
   user_domain_id = default
   auth_type = password
   username = ceilometer
   project_name = service
   password = CEILOMETER_PASS
   interface = internalURL
   region_name = RegionOne
   ```

   Replace CEILOMETER_PASS with the password you chose for the ceilometer user in the Identity service.

Configure Compute to use Telemetry

- Edit the /etc/nova/nova.conf file and configure notifications in the [DEFAULT] section:

```
[DEFAULT]
...
instance_usage_audit = True
instance_usage_audit_period = hour

[notifications]
...
```

(continues on next page)

(continued from previous page)

```
notify_on_state_change = vm_and_task_state

[oslo_messaging_notifications]
...
driver = messagingv2
```

Configure Compute to poll IPMI meters

Note

To enable IPMI meters, ensure IPMITool is installed and the host supports Intel Node Manager.

- Edit the `/etc/sudoers` file and include:

```
ceilometer ALL = (root) NOPASSWD: /usr/bin/ceilometer-rootwrap /etc/
↪ceilometer/rootwrap.conf *
```

- Edit the `/etc/ceilometer/polling.yaml` to include the required meters, for example:

```
- name: ipmi
  interval: 300
  meters:
    - hardware.ipmi.temperature
```

Finalize installation

1. Restart the agent:

```
# service ceilometer-agent-compute restart
# service ceilometer-agent-ipmi restart (optional)
```

2. Restart the Compute service:

```
# service nova-compute restart
```

1.1.4 Verify operation

Verify operation of the Telemetry service. These steps only include the Image service meters to reduce clutter. Environments with ceilometer integration for additional services contain more meters.

Note

Perform these steps on the controller node.

Note

The following uses Gnocchi to verify data. Alternatively, data can be published to a file backend temporarily by using a `file://` publisher.

1. Source the `admin` credentials to gain access to admin-only CLI commands:

```
$ . admin-openrc
```

2. List available resource and its metrics:

```
$ gnocchi resource list  --type image
+--------------------------------------+-------+-------------------------
↪--------+---------+--------------------------------------+-------------
↪-------------------+----------+----------------------------------+-----
↪---------+
| id                                   | type  | project_id              ␣
↪        | user_id | original_resource_id                 | started_at  ␣
↪                        | ended_at | revision_start                  |␣
↪revision_end |
+--------------------------------------+-------+-------------------------
↪--------+---------+--------------------------------------+-------------
↪-------------------+----------+----------------------------------+-----
↪---------+
| a6b387e1-4276-43db-b17a-e10f649d85a3 | image |␣
↪6fd9631226e34531b53814a0f39830a9 | None    | a6b387e1-4276-43db-b17a-
↪e10f649d85a3 | 2017-01-25T23:50:14.423584+00:00 | None     | 2017-01-
↪25T23:50:14.423601+00:00 | None          |
+--------------------------------------+-------+-------------------------
↪--------+---------+--------------------------------------+-------------
↪-------------------+----------+----------------------------------+-----
↪---------+

$ gnocchi resource show a6b387e1-4276-43db-b17a-e10f649d85a3
+-----------------------+------------------------------------------------
↪-----------------+
| Field                 | Value                                          ␣
↪                 |
+-----------------------+------------------------------------------------
↪-----------------+
| created_by_project_id | aca4db3db9904ecc9c1c9bb1763da6a8               ␣
↪                 |
| created_by_user_id    | 07b0945689a4407dbd1ea72c3c5b8d2f               ␣
↪                 |
| creator               |␣
↪07b0945689a4407dbd1ea72c3c5b8d2f:aca4db3db9904ecc9c1c9bb1763da6a8 |
| ended_at              | None                                           ␣
↪                 |
| id                    | a6b387e1-4276-43db-b17a-e10f649d85a3           ␣
↪                 |
| metrics               | image.download: 839afa02-1668-4922-a33e-
```

(continues on next page)

(continued from previous page)

```
↪6b6ea7780715                |
|                       | image.serve: 1132e4a0-9e35-4542-a6ad-
↪d6dc5fb4b835               |
|                       | image.size: 8ecf6c17-98fd-446c-8018-
↪b741dc089a76               |
| original_resource_id  | a6b387e1-4276-43db-b17a-e10f649d85a3                    ␣
↪                    |
| project_id            | 6fd9631226e34531b53814a0f39830a9                        ␣
↪                    |
| revision_end          | None                                                    ␣
↪                    |
| revision_start        | 2017-01-25T23:50:14.423601+00:00                        ␣
↪                    |
| started_at            | 2017-01-25T23:50:14.423584+00:00                        ␣
↪                    |
| type                  | image                                                   ␣
↪                    |
| user_id               | None                                                    ␣
↪                    |
+-----------------------+-------------------------------------------------------
↪-------------------+
```

3. Download the CirrOS image from the Image service:

```
$ IMAGE_ID=$(glance image-list | grep 'cirros' | awk '{ print $2 }')
$ glance image-download $IMAGE_ID > /tmp/cirros.img
```

4. List available meters again to validate detection of the image download:

```
$ gnocchi measures show 839afa02-1668-4922-a33e-6b6ea7780715
+---------------------------+-------------+-----------+
| timestamp                 | granularity |     value |
+---------------------------+-------------+-----------+
| 2017-01-26T15:35:00+00:00 |       300.0 | 3740163.0 |
+---------------------------+-------------+-----------+
```

5. Remove the previously downloaded image file `/tmp/cirros.img`:

```
$ rm /tmp/cirros.img
```

1.1.5 Next steps

Your OpenStack environment now includes the ceilometer service.

To add additional services, see the OpenStack Installation Tutorials and Guides.

This chapter assumes a working setup of OpenStack following the OpenStack Installation Tutorials and Guides.

1.2 Contributor Guide

In the Contributor Guide, you will find documented policies for developing with Ceilometer. This includes the processes we use for bugs, contributor onboarding, core reviewer memberships, and other procedural items.

Ceilometer follows the same workflow as other OpenStack projects. To start contributing to Ceilometer, please follow the workflow found here.

Bug tracker
https://bugs.launchpad.net/ceilometer

Mailing list
http://lists.openstack.org/cgi-bin/mailman/listinfo/openstack-discuss (prefix subjects with `[Ceilometer]` for faster responses)

Wiki
https://wiki.openstack.org/wiki/Ceilometer

Code Hosting
https://opendev.org/openstack/ceilometer/

Code Review
https://review.opendev.org/#/q/status:open+project:openstack/ceilometer,n,z

1.2.1 Overview

Overview

Objectives

The Ceilometer project was started in 2012 with one simple goal in mind: to provide an infrastructure to collect any information needed regarding OpenStack projects. It was designed so that rating engines could use this single source to transform events into billable items which we label as "metering".

As the project started to come to life, collecting an increasing number of meters across multiple projects, the OpenStack community started to realize that a secondary goal could be added to Ceilometer: become a standard way to meter, regardless of the purpose of the collection. This data can then be pushed to any set of targets using provided publishers mentioned in *pipeline-publishers* section.

Metering

If you divide a billing process into a 3 step process, as is commonly done in the telco industry, the steps are:

1. *metering*
2. *rating*
3. *billing*

Ceilometer's initial goal was, and still is, strictly limited to step one. This is a choice made from the beginning not to go into rating or billing, as the variety of possibilities seemed too large for the project to ever deliver a solution that would fit everyone's needs, from private to public clouds. This means that if you are looking at this project to solve your billing needs, this is the right way to go, but certainly not the end of the road for you.

System Architecture

High-Level Architecture

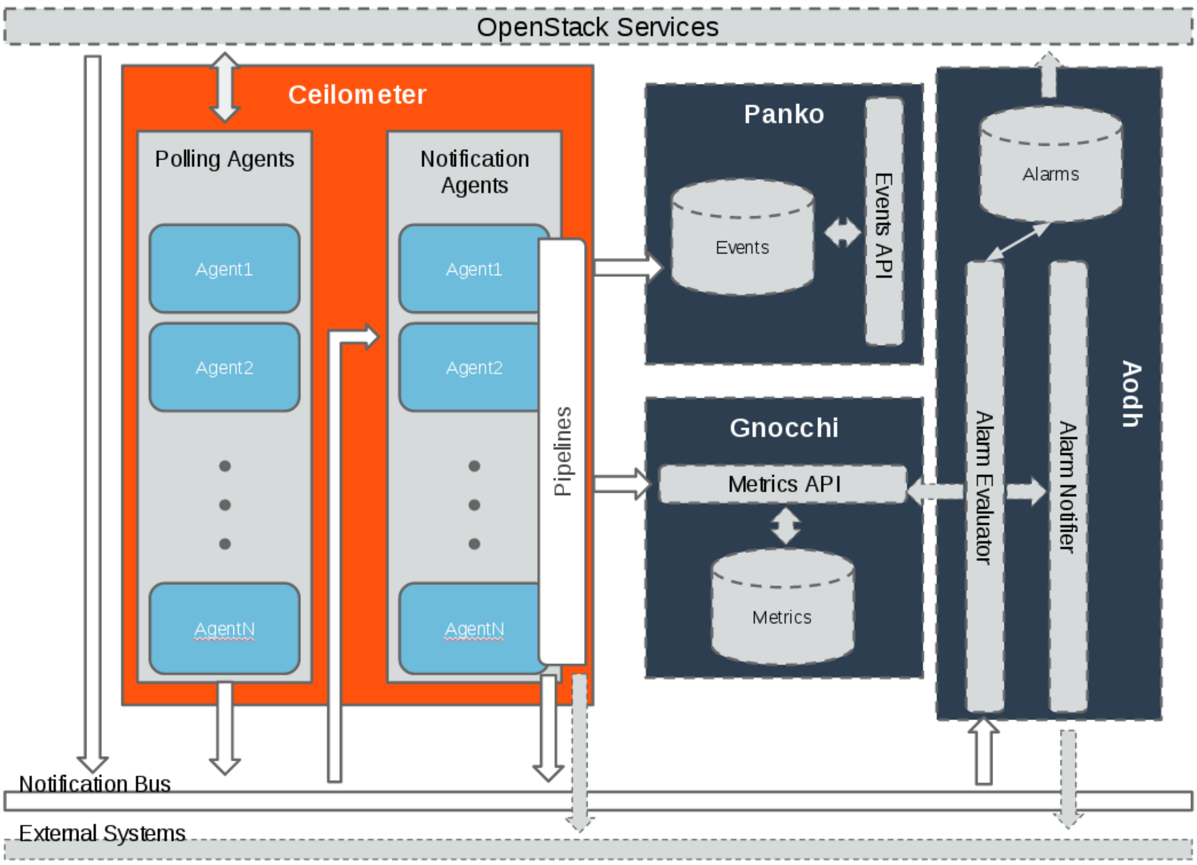

Fig. 1: An overall summary of Ceilometer's logical architecture.

Each of Ceilometer's services are designed to scale horizontally. Additional workers and nodes can be added depending on the expected load. Ceilometer offers two core services:

1. polling agent - daemon designed to poll OpenStack services and build Meters.
2. notification agent - daemon designed to listen to notifications on message queue, convert them to Events and Samples, and apply pipeline actions.

Data normalised and collected by Ceilometer can be sent to various targets. Gnocchi was developed to capture measurement data in a time series format to optimise storage and querying. Gnocchi is intended to replace the existing metering database interface. Additionally, Aodh is the alarming service which can send alerts when user defined rules are broken. Lastly, Panko is the event storage project designed to capture document-oriented data such as logs and system event actions.

Gathering the data

How is data collected?

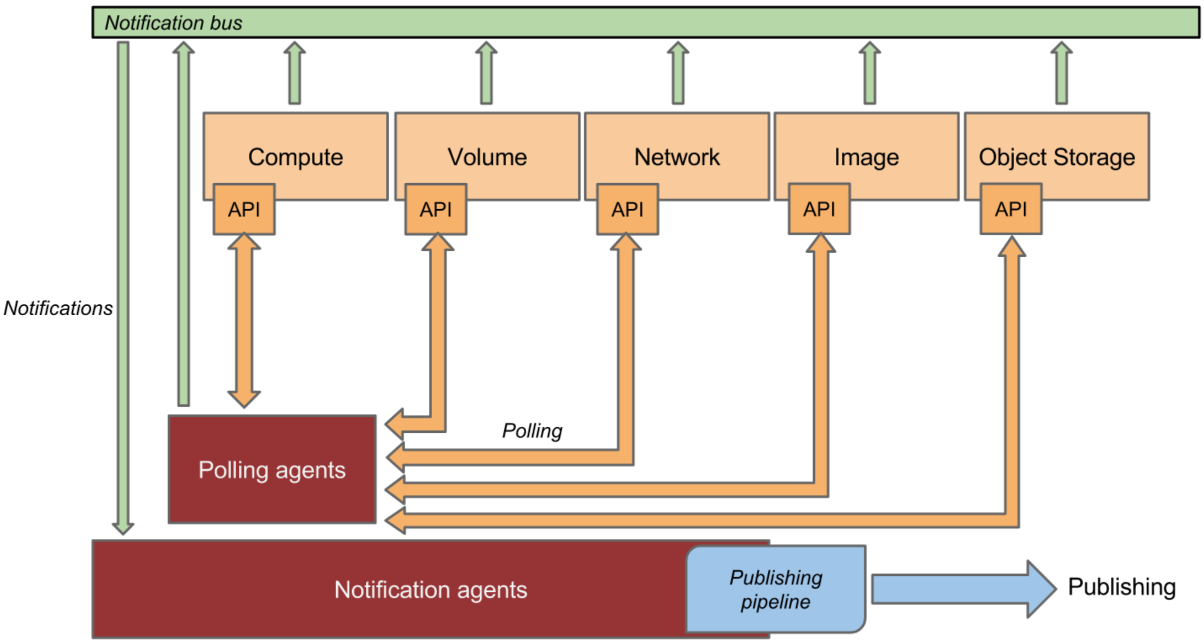

Fig. 2: This is a representation of how the agents gather data from multiple sources.

The Ceilometer project created 2 methods to collect data:

1. *notification agent* which takes messages generated on the notification bus and transforms them into Ceilometer samples or events.
2. *polling agent*, will poll some API or other tool to collect information at a regular interval. The polling approach may impose significant on the API services so should only be used on optimised endpoints.

The first method is supported by the ceilometer-notification agent, which monitors the message queues for notifications. Polling agents can be configured either to poll the local hypervisor or remote APIs (public REST APIs exposed by services and host-level IPMI daemons).

Notification Agent: Listening for data

The heart of the system is the notification daemon (agent-notification) which monitors the message queue for data sent by other OpenStack components such as Nova, Glance, Cinder, Neutron, Swift, Keystone, and Heat, as well as Ceilometer internal communication.

The notification daemon loads one or more *listener* plugins, using the namespace `ceilometer.notification`. Each plugin can listen to any topic, but by default, will listen to `notifications.info`, `notifications.sample`, and `notifications.error`. The listeners grab messages off the configured topics and redistributes them to the appropriate plugins(endpoints) to be processed into Events and Samples.

Sample-oriented plugins provide a method to list the event types they're interested in and a callback for processing messages accordingly. The registered name of the callback is used to enable or disable it using

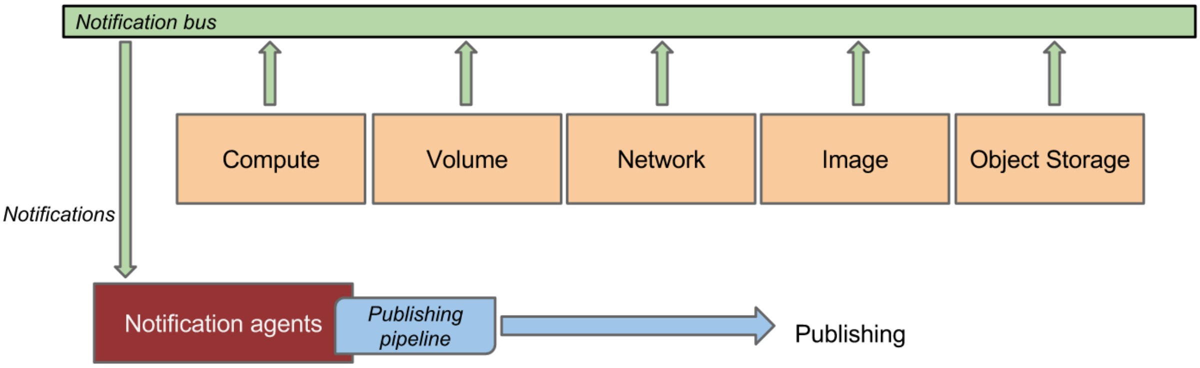

Fig. 3: Notification agent consuming messages from services.

the pipeline of the notification daemon. The incoming messages are filtered based on their event type value before being passed to the callback so the plugin only receives events it has expressed an interest in seeing.

Polling Agent: Asking for data

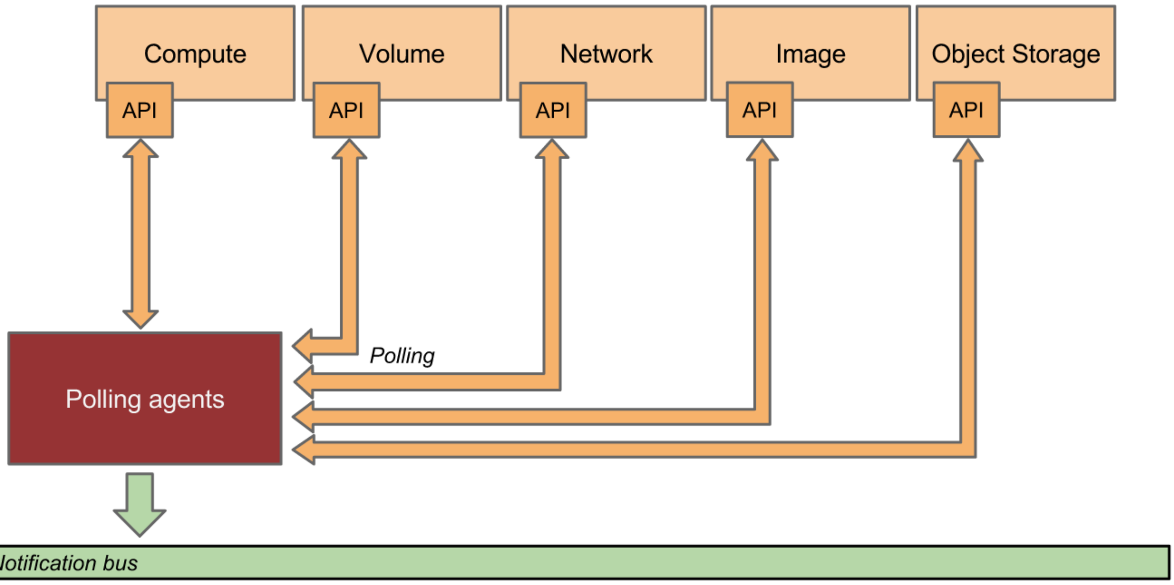

Fig. 4: Polling agent querying services for data.

Polling for compute resources is handled by a polling agent running on the compute node (where communication with the hypervisor is more efficient), often referred to as the compute-agent. Polling via service APIs for non-compute resources is handled by an agent running on a cloud controller node, often referred to the central-agent. A single agent can fulfill both roles in an all-in-one deployment. Conversely, multiple instances of an agent may be deployed, in which case the workload is shared. The polling agent daemon is configured to run one or more *pollster* plugins using any combination of `ceilometer.poll.compute`, `ceilometer.poll.central`, and `ceilometer.poll.ipmi` namespaces

The frequency of polling is controlled via the polling configuration. See *Polling* for details. The agent framework then passes the generated samples to the notification agent for processing.

Processing the data

Pipeline Manager

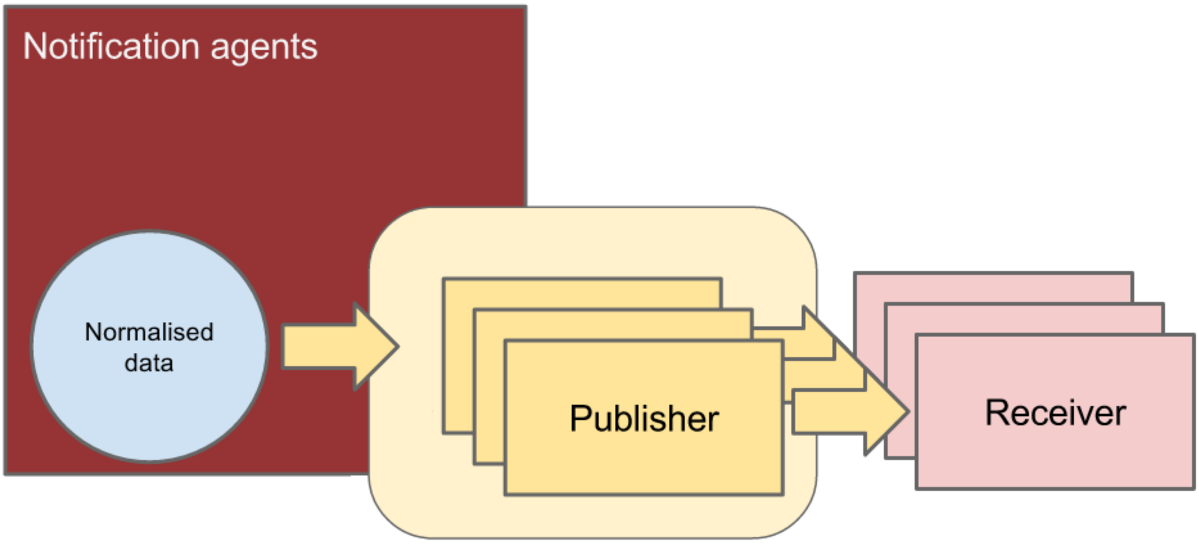

Fig. 5: The assembly of components making the Ceilometer pipeline.

Ceilometer offers the ability to take data gathered by the agents, manipulate it, and publish it in various combinations via multiple pipelines. This functionality is handled by the notification agents.

Publishing the data

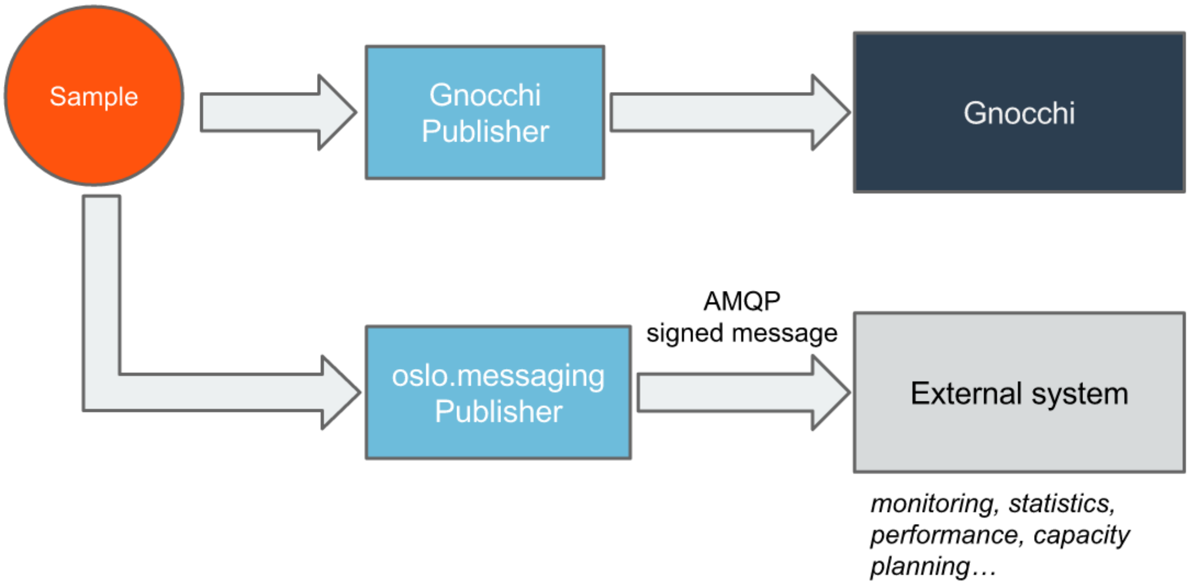

Fig. 6: This figure shows how a sample can be published to multiple destinations.

Currently, processed data can be published using different transport options:

1. gnocchi, which publishes samples/events to Gnocchi API;

2. notifier, a notification based publisher which pushes samples to a message queue which can be consumed by an external system;
3. udp, which publishes samples using UDP packets;
4. http, which targets a REST interface;
5. file, which publishes samples to a file with specified name and location;
6. zaqar, a multi-tenant cloud messaging and notification service for web and mobile developers;
7. https, which is http over SSL and targets a REST interface;
8. prometheus, which publishes samples to Prometheus Pushgateway;

Storing/Accessing the data

Ceilometer is designed solely to generate and normalise cloud data. The data created by Ceilometer can be pushed to any number of target using publishers mentioned in *pipeline-publishers* section. The recommended workflow is to push data to Gnocchi for efficient time-series storage and resource lifecycle tracking.

1.2.2 Data Types

Measurements

Existing meters

For the list of existing meters see the tables under the Measurements page of Ceilometer in the Administrator Guide.

New measurements

Ceilometer is designed to collect measurements from OpenStack services and from other external components. If you would like to add new meters to the currently existing ones, you need to follow the guidelines given in this section.

Types

Three type of meters are defined in Ceilometer:

Type	Definition
Cumulative	Increasing over time (instance hours)
Gauge	Discrete items (floating IPs, image uploads) and fluctuating values (disk I/O)
Delta	Changing over time (bandwidth)

When you're about to add a new meter choose one type from the above list, which is applicable.

Units

1. Whenever a volume is to be measured, SI approved units and their approved symbols or abbreviations should be used. Information units should be expressed in bits ('b') or bytes ('B').
2. For a given meter, the units should NEVER, EVER be changed.
3. When the measurement does not represent a volume, the unit description should always describe WHAT is measured (ie: apples, disk, routers, floating IPs, etc.).
4. When creating a new meter, if another meter exists measuring something similar, the same units and precision should be used.
5. Meters and samples should always document their units in Ceilometer (API and Documentation) and new sampling code should not be merged without the appropriate documentation.

Dimension	Unit	Abbreviations	Note
None	N/A		Dimension-less variable
Volume	byte	B	
Time	seconds	s	

Naming convention

If you plan on adding meters, please follow the convention below:

1. Always use '.' as separator and go from least to most discriminant word. For example, do not use ephemeral_disk_size but disk.ephemeral.size
2. When a part of the name is a variable, it should always be at the end and start with a ':'. For example, do not use <type>.image but image:<type>, where type is your variable name.
3. If you have any hesitation, come and ask in #openstack-telemetry

Meter definitions

Meters definitions by default, are stored in separate configuration file, called `ceilometer/data/meters.d/meters.yaml`. This is essentially a replacement for prior approach of writing notification handlers to consume specific topics.

A detailed description of how to use meter definition is illustrated in the admin_guide.

Events and Event Processing

Events vs. Samples

In addition to Meters, and related Sample data, Ceilometer can also process Events.

While a Sample represents a single numeric datapoint, driving a Meter that represents the changes in that value over time, an Event represents the state of an object in an OpenStack service (such as an Instance in Nova, or an Image in Glance) at a point in time when something of interest has occurred. This can include non-numeric data, such as an instance's flavor, or network address.

In general, Events let you know when something has changed about an object in an OpenStack system, such as the resize of an instance, or creation of an image.

While Samples can be relatively cheap (small), disposable (losing an individual sample datapoint won't matter much), and fast, Events are larger, more informative, and should be handled more consistently (you do not want to lose one).

Event Structure

To facilitate downstream processing (billing and/or aggregation), a *minimum required data set and format <format>* has been defined for services, however events generally contain the following information:

event_type
A dotted string defining what event occurred, such as `compute.instance.resize.start`

message_id
A UUID for this event.

generated
A timestamp of when the event occurred on the source system.

traits
A flat mapping of key-value pairs. The event's Traits contain most of the details of the event. Traits are typed, and can be strings, ints, floats, or datetimes.

raw
(Optional) Mainly for auditing purpose, the full notification message can be stored (unindexed) for future evaluation.

Events from Notifications

Events are primarily created via the notifications system in OpenStack. OpenStack systems, such as Nova, Glance, Neutron, etc. will emit notifications in a JSON format to the message queue when some notable action is taken by that system. Ceilometer will consume such notifications from the message queue, and process them.

The general philosophy of notifications in OpenStack is to emit any and all data someone might need, and let the consumer filter out what they are not interested in. In order to make processing simpler and more efficient, the notifications are stored and processed within Ceilometer as Events. The notification payload, which can be an arbitrarily complex JSON data structure, is converted to a flat set of key-value pairs known as Traits. This conversion is specified by a config file, so that only the specific fields within the notification that are actually needed for processing the event will have to be stored as Traits.

Note that the Event format is meant for efficient processing and querying, there are other means available for archiving notifications (i.e. for audit purposes, etc), possibly to different datastores.

Converting Notifications to Events

In order to make it easier to allow users to extract what they need, the conversion from Notifications to Events is driven by a configuration file (specified by the flag definitions_cfg_file in `ceilometer.conf`).

This includes descriptions of how to map fields in the notification body to Traits, and optional plugins for doing any programmatic translations (splitting a string, forcing case, etc.)

The mapping of notifications to events is defined per event_type, which can be wildcarded. Traits are added to events if the corresponding fields in the notification exist and are non-null. (As a special case, an empty string is considered null for non-text traits. This is due to some openstack projects (mostly Nova) using empty string for null dates.)

If the definitions file is not present, a warning will be logged, but an empty set of definitions will be assumed. By default, any notifications that do not have a corresponding event definition in the definitions file will be converted to events with a set of minimal, default traits. This can be changed by setting the flag drop_unmatched_notifications in the `ceilometer.conf` file. If this is set to True, then any notifications that don't have events defined for them in the file will be dropped. This can be what you want, the notification system is quite chatty by design (notifications philosophy is "tell us everything, we'll ignore what we don't need"), so you may want to ignore the noisier ones if you don't use them.

There is a set of default traits (all are TEXT type) that will be added to all events if the notification has the relevant data:

- service: (All notifications should have this) notification's publisher
- tenant_id
- request_id
- project_id
- user_id

These do not have to be specified in the event definition, they are automatically added, but their definitions can be overridden for a given `event_type`.

Definitions file format

The event definitions file is in YAML format. It consists of a list of event definitions, which are mappings. Order is significant, the list of definitions is scanned in *reverse* order (last definition in the file to the first), to find a definition which matches the notification's event_type. That definition will be used to generate the Event. The reverse ordering is done because it is common to want to have a more general wildcarded definition (such as `compute.instance.*`) with a set of traits common to all of those events, with a few more specific event definitions (like `compute.instance.exists`) afterward that have all of the above traits, plus a few more. This lets you put the general definition first, followed by the specific ones, and use YAML mapping include syntax to avoid copying all of the trait definitions.

Event Definitions

Each event definition is a mapping with two keys (both required):

event_type
This is a list (or a string, which will be taken as a 1 element list) of event_types this definition will handle. These can be wildcarded with unix shell glob syntax. An exclusion listing (starting with a '!') will exclude any types listed from matching. If ONLY exclusions are listed, the definition will match anything not matching the exclusions.

traits
This is a mapping, the keys are the trait names, and the values are trait definitions.

Trait Definitions

Each trait definition is a mapping with the following keys:

type
(optional) The data type for this trait. (as a string). Valid options are: *text*, *int*, *float*, and *datetime*. defaults to *text* if not specified.

fields
A path specification for the field(s) in the notification you wish to extract for this trait. Specifications can be written to match multiple possible fields, the value for the trait will be derived from the matching fields that exist and have a non-null values in the notification. By default the value will be the first such field. (plugins can alter that, if they wish). This is normally a string, but, for convenience, it can be specified as a list of specifications, which will match the fields for all of them. (See *Field Path Specifications* for more info on this syntax.)

plugin
(optional) This is a mapping (For convenience, this value can also be specified as a string, which is interpreted as the name of a plugin to be loaded with no parameters) with the following keys:

name
(string) name of a plugin to load

parameters
(optional) Mapping of keyword arguments to pass to the plugin on initialization. (See documentation on each plugin to see what arguments it accepts.)

Field Path Specifications

The path specifications define which fields in the JSON notification body are extracted to provide the value for a given trait. The paths can be specified with a dot syntax (e.g. `payload.host`). Square bracket syntax (e.g. `payload[host]`) is also supported. In either case, if the key for the field you are looking for contains special characters, like '.', it will need to be quoted (with double or single quotes) like so:

```
payload.image_meta.'org.openstack__1__architecture'
```

The syntax used for the field specification is a variant of JSONPath, and is fairly flexible. (see: https://github.com/kennknowles/python-jsonpath-rw for more info)

Example Definitions file

```yaml
---
- event_type: compute.instance.*
  traits: &instance_traits
    user_id:
      fields: payload.user_id
    instance_id:
      fields: payload.instance_id
    host:
      fields: publisher_id
      plugin:
        name: split
        parameters:
          segment: 1
          max_split: 1
    service_name:
      fields: publisher_id
      plugin: split
    instance_type_id:
      type: int
      fields: payload.instance_type_id
    os_architecture:
      fields: payload.image_meta.'org.openstack__1__architecture'
    launched_at:
      type: datetime
      fields: payload.launched_at
    deleted_at:
      type: datetime
      fields: payload.deleted_at
- event_type:
    - compute.instance.exists
    - compute.instance.update
  traits:
    <<: *instance_traits
    audit_period_beginning:
      type: datetime
      fields: payload.audit_period_beginning
    audit_period_ending:
      type: datetime
      fields: payload.audit_period_ending
```

Trait plugins

Trait plugins can be used to do simple programmatic conversions on the value in a notification field, like splitting a string, lowercasing a value, converting a screwball date into ISO format, or the like. They are initialized with the parameters from the trait definition, if any, which can customize their behavior for a given trait. They are called with a list of all matching fields from the notification, so they can derive a value from multiple fields. The plugin will be called even if there are no fields found matching the field path(s), this lets a plugin set a default value, if needed. A plugin can also reject a value by returning *None*, which will cause the trait not to be added. If the plugin returns anything other than *None*, the trait's value will be set to whatever the plugin returned (coerced to the appropriate type for the trait).

Building Notifications

In general, the payload format OpenStack services emit could be described as the Wild West. The payloads are often arbitrary data dumps at the time of the event which is often susceptible to change. To make consumption easier, the Ceilometer team offers: CADF, an open, cloud standard which helps model cloud events.

1.2.3 Getting Started

Installing development sandbox

In a development environment created by devstack, Ceilometer can be tested alongside other OpenStack services.

Configuring devstack

1. Download devstack.
2. Create a `local.conf` file as input to devstack.
3. The ceilometer services are not enabled by default, so they must be enabled in `local.conf` but adding the following:

   ```
   # Enable the Ceilometer devstack plugin
   enable_plugin ceilometer https://opendev.org/openstack/ceilometer.git
   ```

   By default, all ceilometer services except for ceilometer-ipmi agent will be enabled
4. Enable Gnocchi storage support by including the following in `local.conf`:

   ```
   CEILOMETER_BACKEND=gnocchi
   ```

   Optionally, services which extend Ceilometer can be enabled:

   ```
   enable_plugin aodh https://opendev.org/openstack/aodh
   ```

   These plugins should be added before ceilometer.
5. `./stack.sh`

Running the Tests

Ceilometer includes an extensive set of automated unit tests which are run through tox.

1. Install `tox`:

```
$ sudo pip install tox
```

2. Run the unit and code-style tests:

```
$ cd /opt/stack/ceilometer
$ tox -e py27,pep8
```

As tox is a wrapper around testr, it also accepts the same flags as testr. See the testr documentation for details about these additional flags.

Use a double hyphen to pass options to testr. For example, to run only tests under tests/unit/image:

```
$ tox -e py27 -- image
```

To debug tests (ie. break into pdb debugger), you can use "debug" tox environment. Here's an example, passing the name of a test since you'll normally only want to run the test that hits your breakpoint:

```
$ tox -e debug ceilometer.tests.unit.test_bin
```

For reference, the `debug` tox environment implements the instructions here: https://wiki.openstack.org/wiki/Testr#Debugging_.28pdb.29_Tests

Guru Meditation Reports

Ceilometer contains a mechanism whereby developers and system administrators can generate a report about the state of a running Ceilometer executable. This report is called a *Guru Meditation Report* (*GMR* for short).

Generating a GMR

A *GMR* can be generated by sending the *USR1* signal to any Ceilometer process with support (see below). The *GMR* will then be outputted standard error for that particular process.

For example, suppose that `ceilometer-polling` has process id `8675`, and was run with `2>/var/log/ceilometer/ceilometer-polling.log`. Then, `kill -USR1 8675` will trigger the Guru Meditation report to be printed to `/var/log/ceilometer/ceilometer-polling.log`.

Structure of a GMR

The *GMR* is designed to be extensible; any particular executable may add its own sections. However, the base *GMR* consists of several sections:

Package
Shows information about the package to which this process belongs, including version information

Threads
Shows stack traces and thread ids for each of the threads within this process

Green Threads
Shows stack traces for each of the green threads within this process (green threads don't have thread ids)

Configuration
Lists all the configuration options currently accessible via the CONF object for the current process

Adding Support for GMRs to New Executables

Adding support for a *GMR* to a given executable is fairly easy.

First import the module (currently residing in oslo-incubator), as well as the Ceilometer version module:

```
from oslo_reports import guru_meditation_report as gmr
from ceilometer import version
```

Then, register any additional sections (optional):

```
TextGuruMeditation.register_section('Some Special Section',
                                    some_section_generator)
```

Finally (under main), before running the "main loop" of the executable (usually `service.server(server)` or something similar), register the *GMR* hook:

```
TextGuruMeditation.setup_autorun(version)
```

Extending the GMR

As mentioned above, additional sections can be added to the GMR for a particular executable. For more information, see the inline documentation about oslo.reports: oslo.reports

1.2.4 Development

Writing Agent Plugins

This documentation gives you some clues on how to write a new agent or plugin for Ceilometer if you wish to instrument a measurement which has not yet been covered by an existing plugin.

Plugin Framework

Although we have described a list of the meters Ceilometer should collect, we cannot predict all of the ways deployers will want to measure the resources their customers use. This means that Ceilometer needs to be easy to extend and configure so it can be tuned for each installation. A plugin system based on setuptools entry points makes it easy to add new monitors in the agents. In particular, Ceilometer now uses Stevedore, and you should put your entry point definitions in the `entry_points.txt` file of your Ceilometer egg.

Installing a plugin automatically activates it the next time the ceilometer daemon starts. Rather than running and reporting errors or simply consuming cycles for no-ops, plugins may disable themselves at runtime based on configuration settings defined by other components (for example, the plugin for polling libvirt does not run if it sees that the system is configured using some other virtualization tool). Additionally, if no valid resources can be discovered the plugin will be disabled.

Polling Agents

The polling agent is implemented in `ceilometer/polling/manager.py`. As you will see in the manager, the agent loads all plugins defined in the `ceilometer.poll.*` and `ceilometer.builder.poll.*` namespaces, then periodically calls their `get_samples()` method.

Currently we keep separate namespaces - `ceilometer.poll.compute` and `ceilometer.poll.central` for quick separation of what to poll depending on where is polling agent running. For example, this will load, among others, the `ceilometer.compute.pollsters.instance_stats.CPUPollster`

Pollster

All pollsters are subclasses of `ceilometer.polling.plugin_base.PollsterBase` class. Pollsters must implement one method: `get_samples(self, manager, cache, resources)`, which returns a sequence of `Sample` objects as defined in the `ceilometer/sample.py` file.

Compute plugins are defined as subclasses of the `ceilometer.compute.pollsters.GenericComputePollster` class as defined in the `ceilometer/compute/pollsters/__init__.py` file.

For example, in the `CPUPollster` plugin, the `get_samples` method takes in a given list of resources representing instances on the local host, loops through them and retrieves the *cpu time* details from resource. Similarly, other metrics are built by pulling the appropriate value from the given list of resources.

Notifications

Notifications in OpenStack are consumed by the notification agent and passed through *pipelines* to be normalised and re-published to specified targets.

The existing normalisation pipelines are defined in the namespace `ceilometer.notification.pipeline`.

Each normalisation pipeline are defined as subclass of `ceilometer.pipeline.base.PipelineManager` which interprets and builds pipelines based on a given configuration file. Pipelines are required to define *Source* and *Sink* permutations to describe how to process notification.

Additionally, it must set `get_main_endpoints` which provides endpoints to be added to the main queue listener in the notification agent. This main queue endpoint inherits `ceilometer.pipeline.base.NotificationEndpoint` and defines which notification priorities to listen, normalises the data, and redirects the data for pipeline processing.

Notification endpoints should implement:

`event_types`
A sequence of strings defining the event types the endpoint should handle

`process_notifications(self, priority, notifications)`
Receives an event message from the list provided to `event_types` and returns a sequence of objects. Using the SampleEndpoint, it should yield `Sample` objects as defined in the `ceilometer/sample.py` file.

Two pipeline configurations exist and can be found under `ceilometer.pipeline.*`. The *sample* pipeline loads in multiple endpoints defined in `ceilometer.sample.endpoint` namespace. Each of the endpoints normalises a given notification into different samples.

Ceilometer + Gnocchi Integration

Warning

Remember that custom modification may result in conflicts with upstream upgrades. If not intended to be merged with upstream, it's advisable to directly create resource-types via Gnocchi API.

Managing Resource Types

Resource types in Gnocchi are managed by Ceilometer. The following describes how to add/remove or update Gnocchi resource types to support new Ceilometer data.

The modification or creation of Gnocchi resource type definitions are managed *resources_update_operations* of `ceilometer/gnocchi_client.py`.

The following operations are supported:

1. Adding a new attribute to a resource type. The following adds *flavor_name* attribute to an existing *instance* resource:

```
{"desc": "add flavor_name to instance",
 "type": "update_attribute_type",
 "resource_type": "instance",
 "data": [{
     "op": "add",
     "path": "/attributes/flavor_name",
     "value": {"type": "string", "min_length": 0, "max_length": 255,
               "required": True, "options": {'fill': ''}}
 }]}
```

2. Remove an existing attribute from a resource type. The following removes *server_group* attribute from *instance* resource:

```
{"desc": "remove server_group to instance",
 "type": "update_attribute_type",
 "resource_type": "instance",
 "data": [{
     "op": "remove",
     "path": "/attributes/server_group"
 }]}
```

3. Creating a new resource type. The following creates a new resource type named *nova_compute* with a required attribute *host_name*:

```
{"desc": "add nova_compute resource type",
 "type": "create_resource_type",
 "resource_type": "nova_compute",
 "data": [{
     "attributes": {"host_name": {"type": "string", "min_length": 0,
                    "max_length": 255, "required": True}}
 }]}
```

Note

Do not modify the existing change steps when making changes. Each modification requires a new step to be added and for *ceilometer-upgrade* to be run to apply the change to Gnocchi.

With accomplishing sections above, don't forget to add a new resource type or attributes of a resource type into the `ceilometer/publisher/data/gnocchi_resources.yaml`.

1.3 Administrator Guide

1.3.1 Overview

System architecture

The Telemetry service uses an agent-based architecture. Several modules combine their responsibilities to collect, normalize, and redirect data to be used for use cases such as metering, monitoring, and alerting.

The Telemetry service is built from the following agents:

ceilometer-polling

Polls for different kinds of meter data by using the polling plug-ins (pollsters) registered in different namespaces. It provides a single polling interface across different namespaces.

Note

The `ceilometer-polling` service provides polling support on any namespace but many distributions continue to provide namespace-scoped agents: `ceilometer-agent-central`, `ceilometer-agent-compute`, and `ceilometer-agent-ipmi`.

ceilometer-agent-notification
Consumes AMQP messages from other OpenStack services, normalizes messages, and publishes them to configured targets.

Except for the `ceilometer-polling` agents polling the `compute` or `ipmi` namespaces, all the other services are placed on one or more controller nodes.

The Telemetry architecture depends on the AMQP service both for consuming notifications coming from OpenStack services and internal communication.

Supported databases

The other key external component of Telemetry is the database, where events, samples, alarm definitions, and alarms are stored. Each of the data models have their own storage service and each support various back ends.

The list of supported base back ends for measurements:

- gnocchi

The list of supported base back ends for alarms:

- MySQL
- PostgreSQL

The list of supported base back ends for events:

- ElasticSearch
- MongoDB
- MySQL
- PostgreSQL

Supported hypervisors

The Telemetry service collects information about the virtual machines, which requires close connection to the hypervisor that runs on the compute hosts.

The following is a list of supported hypervisors.

- Libvirt supported hypervisors such as KVM and QEMU
- VMware vSphere

Note

For details about hypervisor support in libvirt please see the Libvirt API support matrix.

Supported networking services

Telemetry is able to retrieve information from external networking services:

- SDN controller meters:
  - OpenDaylight
  - OpenContrail

1.3.2 Configuration

Data collection

The main responsibility of Telemetry in OpenStack is to collect information about the system that can be used by billing systems or interpreted by analytic tooling.

Collected data can be stored in the form of samples or events in the supported databases, which are listed in *Supported databases*.

The available data collection mechanisms are:

Notifications
Processing notifications from other OpenStack services, by consuming messages from the configured message queue system.

Polling
Retrieve information directly from the hypervisor or by using the APIs of other OpenStack services.

Notifications

All OpenStack services send notifications about the executed operations or system state. Several notifications carry information that can be metered. For example, CPU time of a VM instance created by OpenStack Compute service.

The notification agent is responsible for consuming notifications. This component is responsible for consuming from the message bus and transforming notifications into events and measurement samples.

By default, the notification agent is configured to build both events and samples. To enable selective data models, set the required pipelines using *pipelines* option under the *[notification]* section.

Additionally, the notification agent is responsible to send to any supported publisher target such as gnocchi or panko. These services persist the data in configured databases.

The different OpenStack services emit several notifications about the various types of events that happen in the system during normal operation. Not all these notifications are consumed by the Telemetry service, as the intention is only to capture the billable events and notifications that can be used for monitoring or profiling purposes. The notifications handled are contained under the *ceilometer.sample.endpoint* namespace.

Note

Some services require additional configuration to emit the notifications. Please see the *Install and Configure Controller Services* for more details.

Meter definitions

The Telemetry service collects a subset of the meters by filtering notifications emitted by other OpenStack services. You can find the meter definitions in a separate configuration file, called `ceilometer/data/meters.d/meters.yaml`. This enables operators/administrators to add new meters to Telemetry project by updating the `meters.yaml` file without any need for additional code changes.

> **Note**
>
> The `meters.yaml` file should be modified with care. Unless intended, do not remove any existing meter definitions from the file. Also, the collected meters can differ in some cases from what is referenced in the documentation.

It also support loading multiple meter definition files and allow users to add their own meter definitions into several files according to different types of metrics under the directory of `/etc/ceilometer/meters.d`.

A standard meter definition looks like:

```
---
metric:
  - name: 'meter name'
    event_type: 'event name'
    type: 'type of meter eg: gauge, cumulative or delta'
    unit: 'name of unit eg: MB'
    volume: 'path to a measurable value eg: $.payload.size'
    resource_id: 'path to resource id eg: $.payload.id'
    project_id: 'path to project id eg: $.payload.owner'
    metadata: 'addiitonal key-value data describing resource'
```

The definition above shows a simple meter definition with some fields, from which `name`, `event_type`, `type`, `unit`, and `volume` are required. If there is a match on the event type, samples are generated for the meter.

The `meters.yaml` file contains the sample definitions for all the meters that Telemetry is collecting from notifications. The value of each field is specified by using JSON path in order to find the right value from the notification message. In order to be able to specify the right field you need to be aware of the format of the consumed notification. The values that need to be searched in the notification message are set with a JSON path starting with `$.` For instance, if you need the `size` information from the payload you can define it like `$.payload.size`.

A notification message may contain multiple meters. You can use `*` in the meter definition to capture all the meters and generate samples respectively. You can use wild cards as shown in the following example:

```
---
metric:
  - name: $.payload.measurements.[*].metric.[*].name
    event_type: 'event_name.*'
    type: 'delta'
    unit: $.payload.measurements.[*].metric.[*].unit
    volume: payload.measurements.[*].result
    resource_id: $.payload.target
```

(continues on next page)

(continued from previous page)

```
    user_id: $.payload.initiator.id
    project_id: $.payload.initiator.project_id
```

In the above example, the name field is a JSON path with matching a list of meter names defined in the notification message.

You can use complex operations on JSON paths. In the following example, volume and resource_id fields perform an arithmetic and string concatenation:

```
---
metric:
- name: 'compute.node.cpu.idle.percent'
  event_type: 'compute.metrics.update'
  type: 'gauge'
  unit: 'percent'
  volume: payload.metrics[?(@.name='cpu.idle.percent')].value * 100
  resource_id: $.payload.host + "_" + $.payload.nodename
```

You can use the timedelta plug-in to evaluate the difference in seconds between two datetime fields from one notification.

```
---
metric:
- name: 'compute.instance.booting.time'
  event_type: 'compute.instance.create.end'
 type: 'gauge'
 unit: 'sec'
 volume:
   fields: [$.payload.created_at, $.payload.launched_at]
   plugin: 'timedelta'
 project_id: $.payload.tenant_id
 resource_id: $.payload.instance_id
```

Polling

The Telemetry service is intended to store a complex picture of the infrastructure. This goal requires additional information than what is provided by the events and notifications published by each service. Some information is not emitted directly, like resource usage of the VM instances.

Therefore Telemetry uses another method to gather this data by polling the infrastructure including the APIs of the different OpenStack services and other assets, like hypervisors. The latter case requires closer interaction with the compute hosts. To solve this issue, Telemetry uses an agent based architecture to fulfill the requirements against the data collection.

Configuration

Polling rules are defined by the *polling.yaml* file. It defines the pollsters to enable and the interval they should be polled.

Each source configuration encapsulates meter name matching which matches against the entry point of pollster. It also includes: polling interval determination, optional resource enumeration or discovery.

All samples generated by polling are placed on the queue to be handled by the pipeline configuration loaded in the notification agent.

The polling definition may look like the following:

```
---
sources:
  - name: 'source name'
    interval: 'how often the samples should be generated'
    meters:
      - 'meter filter'
    resources:
      - 'list of resource URLs'
    discovery:
      - 'list of discoverers'
```

The *interval* parameter in the sources section defines the cadence of sample generation in seconds.

Polling plugins are invoked according to each source's section whose *meters* parameter matches the plugin's meter name. Its matching logic functions the same as pipeline filtering.

The optional *resources* section of a polling source allows a list of static resource URLs to be configured. An amalgamated list of all statically defined resources are passed to individual pollsters for polling.

The optional *discovery* section of a polling source contains the list of discoverers. These discoverers can be used to dynamically discover the resources to be polled by the pollsters.

If both *resources* and *discovery* are set, the final resources passed to the pollsters will be the combination of the dynamic resources returned by the discoverers and the static resources defined in the *resources* section.

Agents

There are three types of agents supporting the polling mechanism, the `compute agent`, the `central agent`, and the `IPMI agent`. Under the hood, all the types of polling agents are the same `ceilometer-polling` agent, except that they load different polling plug-ins (pollsters) from different namespaces to gather data. The following subsections give further information regarding the architectural and configuration details of these components.

Running **`ceilometer-agent-compute`** is exactly the same as:

```
$ ceilometer-polling --polling-namespaces compute
```

Running **`ceilometer-agent-central`** is exactly the same as:

```
$ ceilometer-polling --polling-namespaces central
```

Running **ceilometer-agent-ipmi** is exactly the same as:

```
$ ceilometer-polling --polling-namespaces ipmi
```

Compute agent

This agent is responsible for collecting resource usage data of VM instances on individual compute nodes within an OpenStack deployment. This mechanism requires a closer interaction with the hypervisor, therefore a separate agent type fulfills the collection of the related meters, which is placed on the host machines to retrieve this information locally.

A Compute agent instance has to be installed on each and every compute node, installation instructions can be found in the *Install and Configure Compute Services* section in the Installation Tutorials and Guides.

The list of supported hypervisors can be found in *Supported hypervisors*. The Compute agent uses the API of the hypervisor installed on the compute hosts. Therefore, the supported meters may be different in case of each virtualization back end, as each inspection tool provides a different set of meters.

The list of collected meters can be found in *OpenStack Compute*. The support column provides the information about which meter is available for each hypervisor supported by the Telemetry service.

Central agent

This agent is responsible for polling public REST APIs to retrieve additional information on OpenStack resources not already surfaced via notifications.

Some of the services polled with this agent are:

- OpenStack Networking
- OpenStack Object Storage
- OpenStack Block Storage

To install and configure this service use the *Install and configure for Red Hat Enterprise Linux and CentOS* section in the Installation Tutorials and Guides.

Although Ceilometer has a set of default polling agents, operators can add new pollsters dynamically via the dynamic pollsters subsystem *Introduction to dynamic pollster subsystem*.

IPMI agent

This agent is responsible for collecting IPMI sensor data and Intel Node Manager data on individual compute nodes within an OpenStack deployment. This agent requires an IPMI capable node with the ipmitool utility installed, which is commonly used for IPMI control on various Linux distributions.

An IPMI agent instance could be installed on each and every compute node with IPMI support, except when the node is managed by the Bare metal service and the `conductor.send_sensor_data` option is set to `true` in the Bare metal service. It is no harm to install this agent on a compute node without IPMI or Intel Node Manager support, as the agent checks for the hardware and if none is available, returns empty data. It is suggested that you install the IPMI agent only on an IPMI capable node for performance reasons.

The list of collected meters can be found in *IPMI meters*.

> **Note**
>
> Do not deploy both the IPMI agent and the Bare metal service on one compute node. If `conductor.send_sensor_data` is set, this misconfiguration causes duplicated IPMI sensor samples.

Data processing and pipelines

The mechanism by which data is processed is called a pipeline. Pipelines, at the configuration level, describe a coupling between sources of data and the corresponding sinks for publication of data. This functionality is handled by the notification agents.

A source is a producer of data: `samples` or `events`. In effect, it is a set of notification handlers emitting datapoints for a set of matching meters and event types.

Each source configuration encapsulates name matching and mapping to one or more sinks for publication.

A sink, on the other hand, is a consumer of data, providing logic for the publication of data emitted from related sources.

In effect, a sink describes a list of one or more publishers.

Pipeline configuration

The notification agent supports two pipelines: one that handles samples and another that handles events. The pipelines can be enabled and disabled by setting *pipelines* option in the *[notifications]* section.

The actual configuration of each pipelines is, by default, stored in separate configuration files: `pipeline.yaml` and `event_pipeline.yaml`. The location of the configuration files can be set by the `pipeline_cfg_file` and `event_pipeline_cfg_file` options listed in *Ceilometer Configuration Options*

The meter pipeline definition looks like:

```
---
sources:
  - name: 'source name'
    meters:
      - 'meter filter'
    sinks:
      - 'sink name'
sinks:
  - name: 'sink name'
    publishers:
      - 'list of publishers'
```

There are several ways to define the list of meters for a pipeline source. The list of valid meters can be found in *Measurements*. There is a possibility to define all the meters, or just included or excluded meters, with which a source should operate:

- To include all meters, use the * wildcard symbol. It is highly advisable to select only the meters that you intend on using to avoid flooding the metering database with unused data.

- To define the list of meters, use either of the following:
  - To define the list of included meters, use the `meter_name` syntax.
  - To define the list of excluded meters, use the `!meter_name` syntax.

Note

The OpenStack Telemetry service does not have any duplication check between pipelines, and if you add a meter to multiple pipelines then it is assumed the duplication is intentional and may be stored multiple times according to the specified sinks.

The above definition methods can be used in the following combinations:

- Use only the wildcard symbol.
- Use the list of included meters.
- Use the list of excluded meters.
- Use wildcard symbol with the list of excluded meters.

Note

At least one of the above variations should be included in the meters section. Included and excluded meters cannot co-exist in the same pipeline. Wildcard and included meters cannot co-exist in the same pipeline definition section.

The publishers section contains the list of publishers, where the samples data should be sent.

Similarly, the event pipeline definition looks like:

```
---
sources:
  - name: 'source name'
    events:
      - 'event filter'
    sinks:
      - 'sink name'
sinks:
  - name: 'sink name'
    publishers:
      - 'list of publishers'
```

The event filter uses the same filtering logic as the meter pipeline.

Publishers

The Telemetry service provides several transport methods to transfer the data collected to an external system. The consumers of this data are widely different, like monitoring systems, for which data loss is acceptable and billing systems, which require reliable data transportation. Telemetry provides methods to fulfill the requirements of both kind of systems.

The publisher component makes it possible to save the data into persistent storage through the message bus or to send it to one or more external consumers. One chain can contain multiple publishers.

To solve this problem, the multi-publisher can be configured for each data point within the Telemetry service, allowing the same technical meter or event to be published multiple times to multiple destinations, each potentially using a different transport.

The following publisher types are supported:

gnocchi (default)

When the gnocchi publisher is enabled, measurement and resource information is pushed to gnocchi for time-series optimized storage. Gnocchi must be registered in the Identity service as Ceilometer discovers the exact path via the Identity service.

More details on how to enable and configure gnocchi can be found on its official documentation page.

prometheus

Metering data can be send to the pushgateway of Prometheus by using:

`prometheus://pushgateway-host:9091/metrics/job/openstack-telemetry`

With this publisher, timestamp are not sent to Prometheus due to Prometheus Pushgateway design. All timestamps are set at the time it scrapes the metrics from the Pushgateway and not when the metric was polled on the OpenStack services.

In order to get timeseries in Prometheus that looks like the reality (but with the lag added by the Prometheus scrapping mechanism). The *scrape_interval* for the pushgateway must be lower and a multiple of the Ceilometer polling interval.

You can read more here

Due to this, this is not recommended to use this publisher for billing purpose as timestamps in Prometheus will not be exact.

notifier

The notifier publisher can be specified in the form of `notifier://?option1=value1&option2=value2`. It emits data over AMQP using oslo.messaging. Any consumer can then subscribe to the published topic for additional processing.

The following customization options are available:

`per_meter_topic`
: The value of this parameter is 1. It is used for publishing the samples on additional `metering_topic.sample_name` topic queue besides the default `metering_topic` queue.

policy
Used for configuring the behavior for the case, when the publisher fails to send the samples, where the possible predefined values are:

default
Used for waiting and blocking until the samples have been sent.

drop
Used for dropping the samples which are failed to be sent.

queue
Used for creating an in-memory queue and retrying to send the samples on the queue in the next samples publishing period (the queue length can be configured with `max_queue_length`, where 1024 is the default value).

topic
The topic name of the queue to publish to. Setting this will override the default topic defined by `metering_topic` and `event_topic` options. This option can be used to support multiple consumers.

udp

This publisher can be specified in the form of `udp://<host>:<port>/`. It emits metering data over UDP.

file

The file publisher can be specified in the form of `file://path?option1=value1&option2=value2`. This publisher records metering data into a file.

> **Note**
>
> If a file name and location is not specified, the `file` publisher does not log any meters, instead it logs a warning message in the configured log file for Telemetry.

The following options are available for the `file` publisher:

max_bytes
When this option is greater than zero, it will cause a rollover. When the specified size is about to be exceeded, the file is closed and a new file is silently opened for output. If its value is zero, rollover never occurs.

backup_count
If this value is non-zero, an extension will be appended to the filename of the old log, as '.1', '.2', and so forth until the specified value is reached. The file that is written and contains the newest data is always the one that is specified without any extensions.

json
If this option is present, will force ceilometer to write json format into the file.

http

The Telemetry service supports sending samples to an external HTTP target. The samples are sent without any modification. To set this option as the notification agents' target, set `http://` as a publisher endpoint in the pipeline definition files. The HTTP target should be set along with the publisher declaration. For example, additional configuration options can be passed in: `http://localhost:80/?option1=value1&option2=value2`

The following options are available:

timeout
The number of seconds before HTTP request times out.

max_retries
The number of times to retry a request before failing.

batch
If false, the publisher will send each sample and event individually, whether or not the notification agent is configured to process in batches.

verify_ssl
If false, the ssl certificate verification is disabled.

The default publisher is `gnocchi`, without any additional options specified. A sample `publishers` section in the `/etc/ceilometer/pipeline.yaml` looks like the following:

```
publishers:
    - gnocchi://
    - udp://10.0.0.2:1234
    - notifier://?policy=drop&max_queue_length=512&topic=custom_target
```

Telemetry best practices

The following are some suggested best practices to follow when deploying and configuring the Telemetry service.

Data collection

1. The Telemetry service collects a continuously growing set of data. Not all the data will be relevant for an administrator to monitor.

    - Based on your needs, you can edit the `polling.yaml` and `pipeline.yaml` configuration files to include select meters to generate or process
    - By default, Telemetry service polls the service APIs every 10 minutes. You can change the polling interval on a per meter basis by editing the `polling.yaml` configuration file.

    Warning

    If the polling interval is too short, it will likely increase the stress on the service APIs.

2. If polling many resources or at a high frequency, you can add additional central and compute agents as necessary. The agents are designed to scale horizontally. For more information refer to the high availability guide.

> **Note**
>
> The High Availability Guide is a work in progress and is changing rapidly while testing continues.

Introduction to dynamic pollster subsystem

The dynamic pollster feature allows system administrators to create/update REST API pollsters on the fly (without changing code). The system reads YAML configures that are found in `pollsters_definitions_dirs` parameter, which has the default at `/etc/ceilometer/pollsters.d`. Operators can use a single file per dynamic pollster or multiple dynamic pollsters per file.

Current limitations of the dynamic pollster system

Currently, the following types of APIs are not supported by the dynamic pollster system:

- Tenant APIs: Tenant APIs are the ones that need to be polled in a tenant fashion. This feature is "a nice" to have, but is currently not implemented.

The dynamic pollsters system configuration (for OpenStack APIs)

Each YAML file in the dynamic pollster feature can use the following attributes to define a dynamic pollster:

> **Warning**
>
> Caution: Ceilometer does not accept complex value data structure for `value` and `metadata` configurations. Therefore, if you are extracting a complex data structure (Object, list, map, or others), you can take advantage of the `Operations on extracted attributes` feature to transform the object into a simple value (string or number)

- `name`: mandatory field. It specifies the name/key of the dynamic pollster. For instance, a pollster for magnum can use the name `dynamic.magnum.cluster`;
- `sample_type`: mandatory field; it defines the sample type. It must be one of the values: `gauge`, `delta`, `cumulative`;
- `unit`: mandatory field; defines the unit of the metric that is being collected. For magnum, for instance, one can use `cluster` as the unit or some other meaningful String value;
- `value_attribute`: mandatory attribute; defines the attribute in the response from the URL of the component being polled. We also accept nested values dictionaries. To use a nested value one can simply use `attribute1.attribute2.<asMuchAsNeeded>.lastattribute`. It is also possible to reference the sample itself using `"." (dot)`; the self reference of the sample is interesting in cases when the attribute might not exist. Therefore, together with the operations options, one can first check if it exist before retrieving it (example: `". | value['some_field']`

if 'some_field' in value else ''"). In our magnum example, we can use status as the value attribute;

- endpoint_type: mandatory field; defines the endpoint type that is used to discover the base URL of the component to be monitored; for magnum, one can use container-infra. Other values are accepted such as volume for cinder endpoints, object-store for swift, and so on;
- url_path: mandatory attribute. It defines the path of the request that we execute on the endpoint to gather data. For example, to gather data from magnum, one can use v1/clusters/detail;
- metadata_fields: optional field. It is a list of all fields that the response of the request executed with url_path that we want to retrieve. To use a nested value one can simply use attribute1.attribute2.<asMuchAsNeeded>.lastattribute. As an example, for magnum, one can use the following values:

```
metadata_fields:
  - "labels"
  - "updated_at"
  - "keypair"
  - "master_flavor_id"
  - "api_address"
  - "master_addresses"
  - "node_count"
  - "docker_volume_size"
  - "master_count"
  - "node_addresses"
  - "status_reason"
  - "coe_version"
  - "cluster_template_id"
  - "name"
  - "stack_id"
  - "created_at"
  - "discovery_url"
  - "container_version"
```

- skip_sample_values: optional field. It defines the values that might come in the value_attribute that we want to ignore. For magnun, one could for instance, ignore some of the status it has for clusters. Therefore, data is not gathered for clusters in the defined status.

```
skip_sample_values:
  - "CREATE_FAILED"
  - "DELETE_FAILED"
```

- value_mapping: optional attribute. It defines a mapping for the values that the dynamic pollster is handling. This is the actual value that is sent to Gnocchi or other backends. If there is no mapping specified, we will use the raw value that is obtained with the use of value_attribute. An example for magnum, one can use:

```
value_mapping:
  CREATE_IN_PROGRESS: "0"
  CREATE_FAILED: "1"
  CREATE_COMPLETE: "2"
  UPDATE_IN_PROGRESS: "3"
```

(continues on next page)

(continued from previous page)

```
  UPDATE_FAILED: "4"
  UPDATE_COMPLETE: "5"
  DELETE_IN_PROGRESS: "6"
  DELETE_FAILED: "7"
  DELETE_COMPLETE: "8"
  RESUME_COMPLETE: "9"
  RESUME_FAILED: "10"
  RESTORE_COMPLETE: "11"
  ROLLBACK_IN_PROGRESS: "12"
  ROLLBACK_FAILED: "13"
  ROLLBACK_COMPLETE: "14"
  SNAPSHOT_COMPLETE: "15"
  CHECK_COMPLETE: "16"
  ADOPT_COMPLETE: "17"
```

- default_value: optional parameter. The default value for the value mapping in case the variable value receives data that is not mapped to something in the value_mapping configuration. This attribute is only used when value_mapping is defined. Moreover, it has a default of -1.
- metadata_mapping: optional parameter. The map used to create new metadata fields. The key is a metadata name that exists in the response of the request we make, and the value of this map is the new desired metadata field that will be created with the content of the metadata that we are mapping. The metadata_mapping can be created as follows:

```
metadata_mapping:
  name: "display_name"
  some_attribute: "new_attribute_name"
```

- preserve_mapped_metadata: optional parameter. It indicates if we preserve the old metadata name when it gets mapped to a new one. The default value is True.
- response_entries_key: optional parameter. This value is used to define the "key" of the response that will be used to look-up the entries used in the dynamic pollster processing. If no response_entries_key is informed by the operator, we will use the first we find. Moreover, if the response contains a list, instead of an object where one of its attributes is a list of entries, we use the list directly. Therefore, this option will be ignored when the API is returning the list/array of entries to be processed directly. We also accept nested values dictionaries. To use a nested value one can simply use attribute1.attribute2.<asMuchAsNeeded>.lastattribute
- user_id_attribute: optional parameter. The default value is user_id. The name of the attribute in the entries that are processed from response_entries_key elements that will be mapped to user_id attribute that is sent to Gnocchi.
- project_id_attribute: optional parameter. The default value is project_id. The name of the attribute in the entries that are processed from response_entries_key elements that will be mapped to project_id attribute that is sent to Gnocchi.
- resource_id_attribute: optional parameter. The default value is id. The name of the attribute in the entries that are processed from response_entries_key elements that will be mapped to id attribute that is sent to Gnocchi.
- headers: optional parameter. It is a map (similar to the metadata_mapping) of key and value that can be used to customize the header of the request that is executed against the URL. This

configuration works for both OpenStack and non-OpenStack dynamic pollster configuration.

```
headers:
  "x-openstack-nova-api-version": "2.46"
```

- `timeout`: optional parameter. Defines the request timeout for the requests executed by the dynamic pollsters to gather data. The default timeout value is 30 seconds. If it is set to *None*, this means that the request never times out on the client side. Therefore, one might have problems if the server never closes the connection. The pollsters are executed serially, one after the other. Therefore, if the request hangs, all pollsters (including the non-dynamic ones) will stop executing.
- `namespaces`: optional parameter. Defines the namespaces (running ceilometer instances) where the pollster will be instantiated. This parameter accepts a single string value or a list of strings. The default value is *central*.

The complete YAML configuration to gather data from Magnum (that has been used as an example) is the following:

```
---

- name: "dynamic.magnum.cluster"
  sample_type: "gauge"
  unit: "cluster"
  value_attribute: "status"
  endpoint_type: "container-infra"
  url_path: "v1/clusters/detail"
  metadata_fields:
    - "labels"
    - "updated_at"
    - "keypair"
    - "master_flavor_id"
    - "api_address"
    - "master_addresses"
    - "node_count"
    - "docker_volume_size"
    - "master_count"
    - "node_addresses"
    - "status_reason"
    - "coe_version"
    - "cluster_template_id"
    - "name"
    - "stack_id"
    - "created_at"
    - "discovery_url"
    - "container_version"
  value_mapping:
    CREATE_IN_PROGRESS: "0"
    CREATE_FAILED: "1"
    CREATE_COMPLETE: "2"
    UPDATE_IN_PROGRESS: "3"
    UPDATE_FAILED: "4"
    UPDATE_COMPLETE: "5"
```

(continues on next page)

(continued from previous page)

```
    DELETE_IN_PROGRESS: "6"
    DELETE_FAILED: "7"
    DELETE_COMPLETE: "8"
    RESUME_COMPLETE: "9"
    RESUME_FAILED: "10"
    RESTORE_COMPLETE: "11"
    ROLLBACK_IN_PROGRESS: "12"
    ROLLBACK_FAILED: "13"
    ROLLBACK_COMPLETE: "14"
    SNAPSHOT_COMPLETE: "15"
    CHECK_COMPLETE: "16"
    ADOPT_COMPLETE: "17"
```

We can also replicate and enhance some hardcoded pollsters. For instance, the pollster to gather VPN connections. Currently, it is always persisting *1* for all of the VPN connections it finds. However, the VPN connection can have multiple statuses, and we should normally only bill for active resources, and not resources on *ERROR* states. An example to gather VPN connections data is the following (this is just an example, and one can adapt and configure as he/she desires):

```
---

- name: "dynamic.network.services.vpn.connection"
  sample_type: "gauge"
  unit: "ipsec_site_connection"
  value_attribute: "status"
  endpoint_type: "network"
  url_path: "v2.0/vpn/ipsec-site-connections"
  metadata_fields:
      - "name"
      - "vpnservice_id"
      - "description"
      - "status"
      - "peer_address"
  value_mapping:
      ACTIVE: "1"
  metadata_mapping:
      name: "display_name"
  default_value: 0
```

- `response_handlers`: optional parameter. Defines the response handlers used to handle the response. For now, the supported values are:

  `json`: This handler will interpret the response as a *JSON* and will convert it to a *dictionary* which can be manipulated using the operations options when mapping the attributes:

  ```
  ---

  - name: "dynamic.json.response"
    sample_type: "gauge"
    [...]
  ```

(continues on next page)

(continued from previous page)

```
  response_handlers:
    - json
```

Response to handle:

```
{
  "test": {
    "list": [1, 2, 3]
  }
}
```

Response handled:

```
{
  'test': {
    'list': [1, 2, 3]
  }
}
```

`xml`: This handler will interpret the response as an *XML* and will convert it to a *dictionary* which can be manipulated using the operations options when mapping the attributes:

```
---

- name: "dynamic.json.response"
  sample_type: "gauge"
  [...]
  response_handlers:
    - xml
```

Response to handle:

```
<test>
  <list>1</list>
  <list>2</list>
  <list>3</list>
</test>
```

Response handled:

```
{
  'test': {
    'list': [1, 2, 3]
  }
}
```

`text`: This handler will interpret the response as a *PlainText* and will convert it to a *dictionary* which can be manipulated using the operations options when mapping the attributes:

```
---
```

(continues on next page)

(continued from previous page)

```
- name: "dynamic.json.response"
  sample_type: "gauge"
  [...]
  response_handlers:
     - text
```

Response to handle:

```
Plain text response
```

Response handled:

```
{
   'out': "Plain text response"
}
```

They can be used together or individually. If not defined, the *default* value will be *json*. If you set 2 or more response handlers, the first configured handler will be used to try to handle the response, if it is not possible, a *DEBUG* log message will be displayed, then the next will be used and so on. If no configured handler was able to handle the response, an empty dict will be returned and a *WARNING* log will be displayed to warn operators that the response was not able to be handled by any configured handler.

The dynamic pollsters system configuration (for non-OpenStack APIs)

The dynamic pollster system can also be used for non-OpenStack APIs. to configure non-OpenStack APIs, one can use all but one attribute of the Dynamic pollster system. The attribute that is not supported is the `endpoint_type`. The dynamic pollster system for non-OpenStack APIs is activated automatically when one uses the configurations `module`.

The extra parameters (in addition to the original ones) that are available when using the Non-OpenStack dynamic pollster sub-subsystem are the following:

- `module`: required parameter. It is the python module name that Ceilometer has to load to use the authentication object when executing requests against the API. For instance, if one wants to create a pollster to gather data from RadosGW, he/she can use the `awsauth` python module.
- `authentication_object`: mandatory parameter. The name of the class that we can find in the `module` that Ceilometer will use as the authentication object in the request. For instance, when using the `awsauth` python module to gather data from RadosGW, one can use the authentication object as `S3Auth`.
- `authentication_parameters`: optional parameter. It is a comma separated value that will be used to instantiate the `authentication_object`. For instance, if we gather data from RadosGW, and we use the `S3Auth` class, the `authentication_parameters` can be configured as `<rados_gw_access_key>, rados_gw_secret_key, rados_gw_host_name`.
- `barbican_secret_id`: optional parameter. The Barbican secret ID, from which, Ceilometer can retrieve the comma separated values of the `authentication_parameters`.

As follows we present an example on how to convert the hard-coded pollster for *radosgw.api.request* metric to the dynamic pollster model:

```yaml
---

- name: "dynamic.radosgw.api.request"
  sample_type: "gauge"
  unit: "request"
  value_attribute: "total.ops"
  url_path: "http://rgw.service.stage.i.ewcs.ch/admin/usage"
  module: "awsauth"
  authentication_object: "S3Auth"
  authentication_parameters: "<access_key>,<secret_key>,<rados_gateway_server>
↪"
  user_id_attribute: "user"
  project_id_attribute: "user"
  resource_id_attribute: "user"
  response_entries_key: "summary"
```

We can take that example a bit further, and instead of gathering the *total .ops* variable, which counts for all the requests (even the unsuccessful ones), we can use the *successful_ops*.

```yaml
---

- name: "dynamic.radosgw.api.request.successful_ops"
  sample_type: "gauge"
  unit: "request"
  value_attribute: "total.successful_ops"
  url_path: "http://rgw.service.stage.i.ewcs.ch/admin/usage"
  module: "awsauth"
  authentication_object: "S3Auth"
  authentication_parameters: "<access_key>, <secret_key>,<rados_gateway_
↪server>"
  user_id_attribute: "user"
  project_id_attribute: "user"
  resource_id_attribute: "user"
  response_entries_key: "summary"
```

The dynamic pollsters system configuration (for local host commands)

The dynamic pollster system can also be used for local host commands, these commands must be installed in the system that is running the Ceilometer compute agent. To configure local hosts commands, one can use all but two attributes of the Dynamic pollster system. The attributes that are not supported are the `endpoint_type` and `url_path`. The dynamic pollster system for local host commands is activated automatically when one uses the configuration `host_command`.

The extra parameter (in addition to the original ones) that is available when using the local host commands dynamic pollster sub-subsystem is the following:

- `host_command`: required parameter. It is the host command that will be executed in the same host the Ceilometer dynamic pollster agent is running. The output of the command will be processed by the pollster and stored in the configured backend.

As follows we present an example on how to use the local host command:

```
---

- name: "dynamic.host.command"
  sample_type: "gauge"
  unit: "request"
  value_attribute: "value"
  response_entries_key: "test"
  host_command: "echo '<test><user_id>id1_u</user_id><project_id>id1_p</
↪project_id><id>id1</id><meta>meta-data-to-store</meta><value>1</value></
↪test>'"
  metadata_fields:
      - "meta"
  response_handlers:
      - xml
```

To execute multi page host commands, the *next_sample_url_attribute* must generate the next sample command, like the following example:

```
---

- name: "dynamic.s3.objects.size"
  sample_type: "gauge"
  unit: "request"
  value_attribute: "Size"
  project_id_attribute: "Owner.ID"
  user_id_attribute: "Owner.ID"
  resource_id_attribute: "Key"
  response_entries_key: "Contents"
  host_command: "aws s3api list-objects"
  next_sample_url_attribute: NextToken | 'aws s3api list-objects --starting-
↪token "' + value + '"'
```

Operations on extracted attributes

The dynamic pollster system can execute Python operations to transform the attributes that are extracted from the JSON response that the system handles.

One example of use case is the RadosGW that uses <project_id$project_id> as the username (which is normally mapped to the Gnocchi resource_id). With this feature (operations on extracted attributes), one can create configurations in the dynamic pollster to clean/normalize that variable. It is as simple as defining *resource_id_attribute: "user | value.split('$')[0].strip()"*

The operations are separated by | symbol. The first element of the expression is the key to be retrieved from the JSON object. The other elements are operations that can be applied to the *value* variable. The value variable is the variable we use to hold the data being extracted. The previous example can be rewritten as: *resource_id_attribute: "user | value.split ('$') | value[0] | value.strip()"*

As follows we present a complete configuration for a RadosGW dynamic pollster that is removing the *$* symbol, and getting the first part of the String.

```
---

- name: "dynamic.radosgw.api.request.successful_ops"
  sample_type: "gauge"
  unit: "request"
  value_attribute: "total.successful_ops"
  url_path: "http://rgw.service.stage.i.ewcs.ch/admin/usage"
  module: "awsauth"
  authentication_object: "S3Auth"
  authentication_parameters: "<access_key>,<secret_key>,<rados_gateway_server>
↪"
  user_id_attribute: "user | value.split ('$') | value[0]"
  project_id_attribute: "user | value.split ('$') | value[0]"
  resource_id_attribute: "user | value.split ('$') | value[0]"
  response_entries_key: "summary"
```

The Dynamic pollster configuration options that support this feature are the following:

- value_attribute
- response_entries_key
- user_id_attribute
- project_id_attribute
- resource_id_attribute

Multi metric dynamic pollsters (handling attribute values with list of objects)

The initial idea for this feature comes from the *categories* fields that we can find in the *summary* object of the RadosGW API. Each user has a *categories* attribute in the response; in the *categories* list, we can find the object that presents in a granular fashion the consumption of different RadosGW API operations such as GET, PUT, POST, and may others.

As follows we present an example of such a JSON response.

```
{
    "entries": [
        {
            "buckets": [
                {
                    "bucket": "",
                    "categories": [
                        {
                            "bytes_received": 0,
                            "bytes_sent": 40,
                            "category": "list_buckets",
                            "ops": 2,
                            "successful_ops": 2
                        }
                    ],
```

(continues on next page)

(continued from previous page)


```
                    "epoch": 1572969600,
                    "owner": "user",
                    "time": "2019-11-21 00:00:00.000000Z"
                },
                {
                    "bucket": "-",
                    "categories": [
                        {
                            "bytes_received": 0,
                            "bytes_sent": 0,
                            "category": "get_obj",
                            "ops": 1,
                            "successful_ops": 0
                        }
                    ],
                    "epoch": 1572969600,
                    "owner": "someOtherUser",
                    "time": "2019-11-21 00:00:00.000000Z"
                }
            ]
        }
    ]
    "summary": [
        {
            "categories": [
                {
                    "bytes_received": 0,
                    "bytes_sent": 0,
                    "category": "create_bucket",
                    "ops": 2,
                    "successful_ops": 2
                },
                {
                    "bytes_received": 0,
                    "bytes_sent": 2120428,
                    "category": "get_obj",
                    "ops": 46,
                    "successful_ops": 46
                },
                {
                    "bytes_received": 0,
                    "bytes_sent": 21484,
                    "category": "list_bucket",
                    "ops": 8,
                    "successful_ops": 8
                },
                {
                    "bytes_received": 6889056,
                    "bytes_sent": 0,
```


(continues on next page)

(continued from previous page)


```
                "category": "put_obj",
                "ops": 46,
                "successful_ops": 46
            }
        ],
        "total": {
            "bytes_received": 6889056,
            "bytes_sent": 2141912,
            "ops": 102,
            "successful_ops": 102
        },
        "user": "user"
    },
    {
        "categories": [
            {
                "bytes_received": 0,
                "bytes_sent": 0,
                "category": "create_bucket",
                "ops": 1,
                "successful_ops": 1
            },
            {
                "bytes_received": 0,
                "bytes_sent": 0,
                "category": "delete_obj",
                "ops": 23,
                "successful_ops": 23
            },
            {
                "bytes_received": 0,
                "bytes_sent": 5371,
                "category": "list_bucket",
                "ops": 2,
                "successful_ops": 2
            },
            {
                "bytes_received": 3444350,
                "bytes_sent": 0,
                "category": "put_obj",
                "ops": 23,
                "successful_ops": 23
            }
        ],
        "total": {
            "bytes_received": 3444350,
            "bytes_sent": 5371,
            "ops": 49,
            "successful_ops": 49
```


(continues on next page)

(continued from previous page)

```
            },
            "user": "someOtherUser"
        }
    ]
}
```

In that context, and having in mind that we have APIs with similar data structures, we developed an extension for the dynamic pollster that enables multi-metric processing for a single pollster. It works as follows.

The pollster name will contain a placeholder for the variable that identifies the "submetric". E.g. *dynamic.radosgw.api.request.{category}*. The placeholder *{category}* indicates the object's attribute that is in the list of objects that we use to load the sub metric name. Then, we must use a special notation in the *value_attribute* configuration to indicate that we are dealing with a list of objects. This is achieved via *[]* (brackets); for instance, in the *dynamic.radosgw.api.request.{category}*, we can use *[categories].ops* as the *value_attribute*. This indicates that the value we retrieve is a list of objects, and when the dynamic pollster processes it, we want it (the pollster) to load the *ops* value for the sub metrics being generated.

Examples on how to create multi-metric pollster to handle data from RadosGW API are presented as follows:

```
---

- name: "dynamic.radosgw.api.request.{category}"
  sample_type: "gauge"
  unit: "request"
  value_attribute: "[categories].ops"
  url_path: "http://rgw.service.stage.i.ewcs.ch/admin/usage"
  module: "awsauth"
  authentication_object: "S3Auth"
  authentication_parameters:  "<access_key>, <secret_key>,<rados_gateway_
↪server>"
  user_id_attribute: "user | value.split('$')[0]"
  project_id_attribute: "user | value.split('$') | value[0]"
  resource_id_attribute: "user  | value.split('$') | value[0]"
  response_entries_key: "summary"

- name: "dynamic.radosgw.api.request.successful_ops.{category}"
  sample_type: "gauge"
  unit: "request"
  value_attribute: "[categories].successful_ops"
  url_path: "http://rgw.service.stage.i.ewcs.ch/admin/usage"
  module: "awsauth"
  authentication_object: "S3Auth"
  authentication_parameters:  "<access_key>, <secret_key>,<rados_gateway_
↪server>"
  user_id_attribute: "user | value.split('$')[0]"
  project_id_attribute: "user | value.split('$') | value[0]"
  resource_id_attribute: "user  | value.split('$') | value[0]"
  response_entries_key: "summary"
```

(continues on next page)

(continued from previous page)

```
- name: "dynamic.radosgw.api.bytes_sent.{category}"
  sample_type: "gauge"
  unit: "request"
  value_attribute: "[categories].bytes_sent"
  url_path: "http://rgw.service.stage.i.ewcs.ch/admin/usage"
  module: "awsauth"
  authentication_object: "S3Auth"
  authentication_parameters:  "<access_key>, <secret_key>,<rados_gateway_
↪server>"
  user_id_attribute: "user | value.split('$')[0]"
  project_id_attribute: "user | value.split('$') | value[0]"
  resource_id_attribute: "user  | value.split('$') | value[0]"
  response_entries_key: "summary"

- name: "dynamic.radosgw.api.bytes_received.{category}"
  sample_type: "gauge"
  unit: "request"
  value_attribute: "[categories].bytes_received"
  url_path: "http://rgw.service.stage.i.ewcs.ch/admin/usage"
  module: "awsauth"
  authentication_object: "S3Auth"
  authentication_parameters:  "<access_key>, <secret_key>,<rados_gateway_
↪server>"
  user_id_attribute: "user | value.split('$')[0]"
  project_id_attribute: "user | value.split('$') | value[0]"
  resource_id_attribute: "user  | value.split('$') | value[0]"
  response_entries_key: "summary"
```

Handling linked API responses

If the consumed API returns a linked response which contains a link to the next response set (page), the Dynamic pollsters can be configured to follow these links and join all linked responses into a single one.

To enable this behavior the operator will need to configure the parameter *next_sample_url_attribute* that must contain a mapper to the response attribute that contains the link to the next response page. This parameter also supports operations like the others *_attribute* dynamic pollster's parameters.

Examples on how to create a pollster to handle linked API responses are presented as follows:

- Example of a simple linked response:
  - API response:

```
{
  "server_link": "http://test.com/v1/test-volumes/marker=c3",
  "servers": [
    {
      "volume": [
        {
          "name": "a",
```

(continues on next page)

(continued from previous page)

```
          "tmp": "ra"
        }
      ],
      "id": 1,
      "name": "a1"
    },
    {
      "volume": [
        {
          "name": "b",
          "tmp": "rb"
        }
      ],
      "id": 2,
      "name": "b2"
    },
    {
      "volume": [
        {
          "name": "c",
          "tmp": "rc"
        }
      ],
      "id": 3,
      "name": "c3"
    }
  ]
}
```

- Pollster configuration:

```
---

- name: "dynamic.linked.response"
  sample_type: "gauge"
  unit: "request"
  value_attribute: "[volume].tmp"
  url_path: "v1/test-volumes"
  response_entries_key: "servers"
  next_sample_url_attribute: "server_link"
```

- Example of a complex linked response:
  - API response:

```
{
  "server_link": [
    {
      "href": "http://test.com/v1/test-volumes/marker=c3",
      "rel": "next"
```

(continues on next page)

(continued from previous page)

```
    },
    {
      "href": "http://test.com/v1/test-volumes/marker=b1",
      "rel": "prev"
    }
  ],
  "servers": [
    {
      "volume": [
        {
          "name": "a",
          "tmp": "ra"
        }
      ],
      "id": 1,
      "name": "a1"
    },
    {
      "volume": [
        {
          "name": "b",
          "tmp": "rb"
        }
      ],
      "id": 2,
      "name": "b2"
    },
    {
      "volume": [
        {
          "name": "c",
          "tmp": "rc"
        }
      ],
      "id": 3,
      "name": "c3"
    }
  ]
}
```

– Pollster configuration:

```
---

- name: "dynamic.linked.response"
  sample_type: "gauge"
  unit: "request"
  value_attribute: "[volume].tmp"
  url_path: "v1/test-volumes"
  response_entries_key: "servers"
```

(continues on next page)

(continued from previous page)

```
  next_sample_url_attribute: "server_link | filter(lambda v: v.
↪get('rel') == 'next', value) | list(value) | value[0] | value.
↪get('href')"
```

OpenStack Dynamic pollsters metadata enrichment with other OpenStack API's data

Sometimes we want/need to add/gather extra metadata for the samples being handled by Ceilometer Dynamic pollsters, such as the project name, domain id, domain name, and other metadata that are not always accessible via the OpenStack component where the sample is gathered.

For instance, when gathering the status of virtual machines (VMs) from Nova, we only have the *tenant_id*, which must be used as the *project_id*. However, for billing and later invoicing one might need/want the project name, domain id, and other metadata that are available in Keystone (and maybe some others that are scattered over other components). To achieve that, one can use the OpenStack metadata enrichment option. As follows we present an example that shows a dynamic pollster configuration to gather virtual machine (VM) status, and to enrich the data pushed to the storage backend (e.g. Gnocchi) with project name, domain ID, and domain name.

```
---

- name: "dynamic_pollster.instance.status"
  next_sample_url_attribute: "server_links | filter(lambda v: v.get(
↪'rel') == 'next', value) | list(value) | value[0] | value.get('href
↪') | value.replace('http:', 'https:')"
  sample_type: "gauge"
  unit: "server"
  value_attribute: "status"
  endpoint_type: "compute"
  url_path: "/v2.1/servers/detail?all_tenants=true"
  headers:
    "Openstack-API-Version": "compute 2.65"
  project_id_attribute: "tenant_id"
  metadata_fields:
    - "status"
    - "name"
    - "flavor.vcpus"
    - "flavor.ram"
    - "flavor.disk"
    - "flavor.ephemeral"
    - "flavor.swap"
    - "flavor.original_name"
    - "image | value or { 'id': '' } | value['id']"
    - "OS-EXT-AZ:availability_zone"
    - "OS-EXT-SRV-ATTR:host"
    - "user_id"
    - "tags | ','.join(value)"
    - "locked"
  value_mapping:
    ACTIVE: "1"
```

(continues on next page)

(continued from previous page)

```
default_value: 0
metadata_mapping:
  "OS-EXT-AZ:availability_zone": "dynamic_availability_zone"
  "OS-EXT-SRV-ATTR:host": "dynamic_host"
  "flavor.original_name": "dynamic_flavor_name"
  "flavor.vcpus": "dynamic_flavor_vcpus"
  "flavor.ram": "dynamic_flavor_ram"
  "flavor.disk": "dynamic_flavor_disk"
  "flavor.ephemeral": "dynamic_flavor_ephemeral"
  "flavor.swap": "dynamic_flavor_swap"
  "image | value or { 'id': '' } | value['id']": "dynamic_image_ref
↪"
  "name": "dynamic_display_name"
  "locked": "dynamic_locked"
  "tags | ','.join(value)": "dynamic_tags"
extra_metadata_fields_cache_seconds: 3600
extra_metadata_fields_skip:
  - value: '1'
    metadata:
      dynamic_flavor_vcpus: 4
  - value: '1'
    metadata:
      dynamic_flavor_vcpus: 2
extra_metadata_fields:
  - name: "project_name"
    endpoint_type: "identity"
    url_path: "'/v3/projects/' + str(sample['project_id'])"
    headers:
      "Openstack-API-Version": "identity latest"
    value: "name"
    extra_metadata_fields_cache_seconds: 1800 # overriding the␣
↪default cache policy
    metadata_fields:
        - id
  - name: "domain_id"
    endpoint_type: "identity"
    url_path: "'/v3/projects/' + str(sample['project_id'])"
    headers:
      "Openstack-API-Version": "identity latest"
    value: "domain_id"
    metadata_fields:
        - id
  - name: "domain_name"
    endpoint_type: "identity"
    url_path: "'/v3/domains/' + str(extra_metadata_captured[
↪'domain_id'])"
    headers:
      "Openstack-API-Version": "identity latest"
    value: "name"
```

(continues on next page)

(continued from previous page)

```
    metadata_fields:
        - id
  - name: "operating-system"
    host_command: "'get-vm --vm-name ' + str(extra_metadata_by_
↪name['project_name']['metadata']['id'])"
    value: "os"
```

The above example can be used to gather and persist in the backend the status of VMs. It will persist *1* in the backend as a measure for every collecting period if the VM's status is *ACTIVE*, and *0* otherwise. This is quite useful to create hashmap rating rules for running VMs in CloudKitty. Then, to enrich the resource in the storage backend, we are adding extra metadata that are collected in Keystone and in the local host via the *extra_metadata_fields* options. If you have multiples *extra_metadata_fields* defining the same *metadata_field*, the last not *None* metadata value will be used.

To operate values in the *extra_metadata_fields*, you can access 3 local variables:

- `sample`: it is a dictionary which holds the current data of the root sample. The root sample is the final sample that will be persisted in the configured storage backend.
- `extra_metadata_captured`: it is a dictionary which holds the current data of all *extra_metadata_fields* processed before this one. If you have multiples *extra_metadata_fields* defining the same *metadata_field*, the last not *None* metadata value will be used.
- `extra_metadata_by_name`: it is a dictionary which holds the data of all *extra_metadata_fields* processed before this one. No data is overwritten in this variable. To access an specific *extra_metadata_field* using this variable, you can do *extra_metadata_by_name['<extra_metadata_field_name>']['value']* to get its value, or *extra_metadata_by_name['<extra_metadata_field_name>']['metadata']['<metadata>']* to get its metadata.

The metadata enrichment feature has the following options:

- `extra_metadata_fields_cache_seconds`: optional parameter. Defines the extra metadata request's response cache. Some requests, such as the ones executed against Keystone to retrieve extra metadata are rather static. Therefore, one does not need to constantly re-execute the request. That is the reason why we cache the response of such requests. By default the cache time to live (TTL) for responses is *3600* seconds. However, this value can be increased of decreased.
- `extra_metadata_fields`: optional parameter. This option is a list of objects or a single one, where each one of its elements is an dynamic pollster configuration set. Each one of the extra metadata definition can have the same options defined in the dynamic pollsters, including the *extra_metadata_fields* option, so this option is a multi-level option. When defined, the result of the collected data will be merged in the final sample resource metadata. If some of the required dynamic pollster configuration is not set in the *extra_metadata_fields*, will be used the parent pollster configuration, except the *name*.
- `extra_metadata_fields_skip`: optional parameter. This option is a list of objects or a single one, where each one of its elements is a set of key/value pairs. When defined, if any set of key/value pairs is a subset of the collected sample, then the extra_metadata_fields gathering of this sample will be skipped.

1.3.3 Data Types

Measurements

The Telemetry service collects meters within an OpenStack deployment. This section provides a brief summary about meters format and origin and also contains the list of available meters.

Telemetry collects meters by polling the infrastructure elements and also by consuming the notifications emitted by other OpenStack services. For more information about the polling mechanism and notifications see *Data collection*. There are several meters which are collected by polling and by consuming. The origin for each meter is listed in the tables below.

Note

You may need to configure Telemetry or other OpenStack services in order to be able to collect all the samples you need. For further information about configuration requirements see the Telemetry chapter in the Installation Tutorials and Guides.

Telemetry uses the following meter types:

Type	Description
Cumulative	Increasing over time (instance hours)
Delta	Changing over time (bandwidth)
Gauge	Discrete items (floating IPs, image uploads) and fluctuating values (disk I/O)

Telemetry provides the possibility to store metadata for samples. This metadata can be extended for OpenStack Compute and OpenStack Object Storage.

In order to add additional metadata information to OpenStack Compute you have two options to choose from. The first one is to specify them when you boot up a new instance. The additional information will be stored with the sample in the form of `resource_metadata.user_metadata.*`. The new field should be defined by using the prefix `metering.`. The modified boot command look like the following:

```
$ openstack server create --property metering.custom_metadata=a_value my_vm
```

The other option is to set the `reserved_metadata_keys` to the list of metadata keys that you would like to be included in `resource_metadata` of the instance related samples that are collected for OpenStack Compute. This option is included in the `DEFAULT` section of the `ceilometer.conf` configuration file.

You might also specify headers whose values will be stored along with the sample data of OpenStack Object Storage. The additional information is also stored under `resource_metadata`. The format of the new field is `resource_metadata.http_header_$name`, where `$name` is the name of the header with `-` replaced by `_`.

For specifying the new header, you need to set `metadata_headers` option under the `[filter:ceilometer]` section in `proxy-server.conf` under the `swift` folder. You can use this additional data for instance to distinguish external and internal users.

Measurements are grouped by services which are polled by Telemetry or emit notifications that this service consumes.

OpenStack Compute

The following meters are collected for OpenStack Compute.

Name	Type	Unit	Resource	Origin	Support	Note
Meters added in the Mitaka release or earlier						
memory	Gauge	MB	instance ID	Notification	Libvirt	Volu
memory.usage	Gauge	MB	instance ID	Pollster	Libvirt, vSphere,	Volu
memory.resident	Gauge	MB	instance ID	Pollster	Libvirt	Volu
cpu	Cumulative	ns	instance ID	Pollster	Libvirt	CPU
vcpus	Gauge	vcpu	instance ID	Notification	Libvirt	Num
disk.device.read.requests	Cumulative	request	disk ID	Pollster	Libvirt	Num
disk.device.write.requests	Cumulative	request	disk ID	Pollster	Libvirt	Num
disk.device.read.bytes	Cumulative	B	disk ID	Pollster	Libvirt	Volu
disk.device.write.bytes	Cumulative	B	disk ID	Pollster	Libvirt	Volu
disk.root.size	Gauge	GB	instance ID	Notification	Libvirt	Size
disk.ephemeral.size	Gauge	GB	instance ID	Notification	Libvirt	Size
disk.device.capacity	Gauge	B	disk ID	Pollster	Libvirt	The a
disk.device.allocation	Gauge	B	disk ID	Pollster	Libvirt	The a
disk.device.usage	Gauge	B	disk ID	Pollster	Libvirt	The p
network.incoming.bytes	Cumulative	B	interface ID	Pollster	Libvirt	Num
network.outgoing.bytes	Cumulative	B	interface ID	Pollster	Libvirt	Num
network.incoming.packets	Cumulative	packet	interface ID	Pollster	Libvirt	Num
network.outgoing.packets	Cumulative	packet	interface ID	Pollster	Libvirt	Num
Meters added in the Newton release						
cpu_l3_cache	Gauge	B	instance ID	Pollster	Libvirt	L3 ca
memory.bandwidth.total	Gauge	B/s	instance ID	Pollster	Libvirt	Total
memory.bandwidth.local	Gauge	B/s	instance ID	Pollster	Libvirt	Band
perf.cpu.cycles	Gauge	cycle	instance ID	Pollster	Libvirt	the n
perf.instructions	Gauge	instruction	instance ID	Pollster	Libvirt	the c
perf.cache.references	Gauge	count	instance ID	Pollster	Libvirt	the c
perf.cache.misses	Gauge	count	instance ID	Pollster	Libvirt	the c
Meters added in the Ocata release						
network.incoming.packets.drop	Cumulative	packet	interface ID	Pollster	Libvirt	Num
network.outgoing.packets.drop	Cumulative	packet	interface ID	Pollster	Libvirt	Num
network.incoming.packets.error	Cumulative	packet	interface ID	Pollster	Libvirt	Num
network.outgoing.packets.error	Cumulative	packet	interface ID	Pollster	Libvirt	Num
Meters added in the Pike release						
memory.swap.in	Cumulative	MB	instance ID	Pollster	Libvirt	Mem
memory.swap.out	Cumulative	MB	instance ID	Pollster	Libvirt	Mem
Meters added in the Queens release						
disk.device.read.latency	Cumulative	ns	Disk ID	Pollster	Libvirt	Total
disk.device.write.latency	Cumulative	ns	Disk ID	Pollster	Libvirt	Total

Note

To enable the libvirt `memory.usage` support, you need to install libvirt version 1.1.1+, QEMU version 1.5+, and you also need to prepare suitable balloon driver in the image. It is applicable particularly for Windows guests, most modern Linux distributions already have it built in. Telemetry is not able to fetch the `memory.usage` samples without the image balloon driver.

Note

To enable libvirt `disk.*` support when running on RBD-backed shared storage, you need to install libvirt version 1.2.16+.

OpenStack Compute is capable of collecting CPU related meters from the compute host machines. In order to use that you need to set the `compute_monitors` option to `cpu.virt_driver` in the `nova.conf` configuration file. For further information see the Compute configuration section in the Compute chapter of the OpenStack Configuration Reference.

The following host machine related meters are collected for OpenStack Compute:

Name	Type	Unit	Resource	Origin	Note
Meters added in the Mitaka release or earlier					
compute.node.cpu.frequency	Gauge	MHz	host ID	Notification	CPU frequency
compute.node.cpu.kernel.time	Cumulative	ns	host ID	Notification	CPU kernel time
compute.node.cpu.idle.time	Cumulative	ns	host ID	Notification	CPU idle time
compute.node.cpu.user.time	Cumulative	ns	host ID	Notification	CPU user mode time
compute.node.cpu.iowait.time	Cumulative	ns	host ID	Notification	CPU I/O wait time
compute.node.cpu.kernel.percent	Gauge	%	host ID	Notification	CPU kernel percentage
compute.node.cpu.idle.percent	Gauge	%	host ID	Notification	CPU idle percentage
compute.node.cpu.user.percent	Gauge	%	host ID	Notification	CPU user mode percentage
compute.node.cpu.iowait.percent	Gauge	%	host ID	Notification	CPU I/O wait percentage
compute.node.cpu.percent	Gauge	%	host ID	Notification	CPU utilization

IPMI meters

Telemetry captures notifications that are emitted by the Bare metal service. The source of the notifications are IPMI sensors that collect data from the host machine.

Alternatively, IPMI meters can be generated by deploying the ceilometer-agent-ipmi on each IPMI-capable node. For further information about the IPMI agent see *IPMI agent*.

Warning

To avoid duplication of metering data and unnecessary load on the IPMI interface, do not deploy the IPMI agent on nodes that are managed by the Bare metal service and keep the `conductor.send_sensor_data` option set to `False` in the `ironic.conf` configuration file.

The following IPMI sensor meters are recorded:

Name	Type	Unit	Resource	Origin	Note
Meters added in the Mitaka release or earlier					
hardware.ipmi.fan	Gauge	RPM	fan sensor	Notification, Pollster	Fan rounds per minute (RPM)
hardware.ipmi.temperatur	Gauge	C	temperature sensor	Notification, Pollster	Temperature reading from sensor
hardware.ipmi.current	Gauge	A	current sensor	Notification, Pollster	Current reading from sensor
hardware.ipmi.voltage	Gauge	V	voltage sensor	Notification, Pollster	Voltage reading from sensor

Note

The sensor data is not available in the Bare metal service by default. To enable the meters and configure this module to emit notifications about the measured values see the Installation Guide for the Bare metal service.

Besides generic IPMI sensor data, the following Intel Node Manager meters are recorded from capable platform:

Name	Type	Unit	Resource	Origin	Note
Meters added in the Mitaka release or earlier					
hardware.ipmi.node.power	Gauge	W	host ID	Pollster	Current power of the system
hardware.ipmi.node.temperatu	Gauge	C	host ID	Pollster	Current temperature of the system
hardware.ipmi.node.inlet_temp	Gauge	C	host ID	Pollster	Inlet temperature of the system
hardware.ipmi.node.outlet_tem	Gauge	C	host ID	Pollster	Outlet temperature of the system
hardware.ipmi.node.airflow	Gauge	CFM	host ID	Pollster	Volumetric airflow of the system, expressed as 1/10th of CFM
hardware.ipmi.node.cups	Gauge	CUPS	host ID	Pollster	CUPS(Compute Usage Per Second) index data of the system
hardware.ipmi.node.cpu_util	Gauge	%	host ID	Pollster	CPU CUPS utilization of the system
hardware.ipmi.node.mem_util	Gauge	%	host ID	Pollster	Memory CUPS utilization of the system
hardware.ipmi.node.io_util	Gauge	%	host ID	Pollster	IO CUPS utilization of the system

OpenStack Image service

The following meters are collected for OpenStack Image service:

Name	Type	Unit	Resource	Origin	Note
Meters added in the Mitaka release or earlier					
image.size	Gauge	B	image ID	Notification, Pollster	Size of the uploaded image
image.download	Delta	B	image ID	Notification	Image is downloaded
image.serve	Delta	B	image ID	Notification	Image is served out

OpenStack Block Storage

The following meters are collected for OpenStack Block Storage:

Name	Type	Unit	Resource	Origin	Note
Meters added in the Mitaka release or earlier					
volume.size	Gauge	GB	volume ID	Notification	Size of the volume
snapshot.size	Gauge	GB	snapshot ID	Notification	Size of the snapshot
Meters added in the Queens release					
volume.provider.capacity.total	Gauge	GB	hostname	Notification	Total volume capacity on host
volume.provider.capacity.free	Gauge	GB	hostname	Notification	Free volume capacity on host
volume.provider.capacity.allocate	Gauge	GB	hostname	Notification	Assigned volume capacity on host by Cinder
volume.provider.capacity.provisi	Gauge	GB	hostname	Notification	Assigned volume capacity on host
volume.provider.capacity.virtual_	Gauge	GB	hostname	Notification	Virtual free volume capacity on host
volume.provider.pool.capacity.to	Gauge	GB	hostname#pool	Notification	Total volume capacity in pool
volume.provider.pool.capacity.fr	Gauge	GB	hostname#pool	Notification	Free volume capacity in pool
volume.provider.pool.capacity.al	Gauge	GB	hostname#pool	Notification	Assigned volume capacity in pool by Cinder
volume.provider.pool.capacity.pr	Gauge	GB	hostname#pool	Notification	Assigned volume capacity in pool
volume.provider.pool.capacity.vi	Gauge	GB	hostname#pool	Notification	Virtual free volume capacity in pool

OpenStack File Share

The following meters are collected for OpenStack File Share:

Name	Type	Unit	Resource	Origin	Note
Meters added in the Pike release					
manila.share.size	Gauge	GB	share ID	Notification	Size of the file share

OpenStack Object Storage

The following meters are collected for OpenStack Object Storage:

Name	Type	Unit	Resource	Origin	Note
Meters added in the Mitaka release or earlier					
storage.objects	Gauge	object	storage ID	Pollster	Number of objects
storage.objects.size	Gauge	B	storage ID	Pollster	Total size of stored objects
storage.objects.containers	Gauge	container	storage ID	Pollster	Number of containers
storage.objects.incoming.by	Delta	B	storage ID	Notification	Number of incoming bytes
storage.objects.outgoing.by	Delta	B	storage ID	Notification	Number of outgoing bytes
storage.containers.objects	Gauge	object	storage ID/container	Pollster	Number of objects in container
storage.containers.objects.s	Gauge	B	storage ID/container	Pollster	Total size of stored objects in container

Ceph Object Storage

In order to gather meters from Ceph, you have to install and configure the Ceph Object Gateway (radosgw) as it is described in the Installation Manual. You also have to enable usage logging in order to get the related meters from Ceph. You will need an `admin` user with `users`, `buckets`, `metadata` and `usage caps` configured.

In order to access Ceph from Telemetry, you need to specify a `service group` for `radosgw` in the `ceilometer.conf` configuration file along with `access_key` and `secret_key` of the `admin` user mentioned above.

The following meters are collected for Ceph Object Storage:

Name	Type	Unit	Resource	Origin	Note
Meters added in the Mitaka release or earlier					
radosgw.objects	Gauge	object	storage ID	Pollster	Number of objects
radosgw.objects.size	Gauge	B	storage ID	Pollster	Total size of stored objects
radosgw.objects.contai	Gauge	container	storage ID	Pollster	Number of containers
radosgw.api.request	Gauge	request	storage ID	Pollster	Number of API requests against Ceph Object Gateway (radosgw)
radosgw.containers.ob	Gauge	object	storage ID/container	Pollster	Number of objects in container
radosgw.containers.ob	Gauge	B	storage ID/container	Pollster	Total size of stored objects in container

Note

The `usage` related information may not be updated right after an upload or download, because the Ceph Object Gateway needs time to update the usage properties. For instance, the default configuration needs approximately 30 minutes to generate the usage logs.

OpenStack Identity

The following meters are collected for OpenStack Identity:

Name	Type	Unit	Resource	Origin	Note
Meters added in the Mitaka release or earlier					
identity.authenticate.success	Delta	user	user ID	Notification	User successfully authenticated
identity.authenticate.pending	Delta	user	user ID	Notification	User pending authentication
identity.authenticate.failure	Delta	user	user ID	Notification	User failed to authenticate

OpenStack Networking

The following meters are collected for OpenStack Networking:

Name	Type	Unit	Resource	Origin	Note
Meters added in the Mitaka release or earlier					
bandwidth	Delta	B	label ID	Notification	Bytes through this l3 metering label

SDN controllers

The following meters are collected for SDN:

Name	Type	Unit	Resource	Origin	Note
Meters added in the Mitaka release or earlier					
switch	Gauge	switch	switch ID	Pollster	Existence of switch
switch.port	Gauge	port	switch ID	Pollster	Existence of port
switch.port.receive.packets	Cumulative	packet	switch ID	Pollster	Packets received on port
switch.port.transmit.packets	Cumulative	packet	switch ID	Pollster	Packets transmitted on port
switch.port.receive.bytes	Cumulative	B	switch ID	Pollster	Bytes received on port
switch.port.transmit.bytes	Cumulative	B	switch ID	Pollster	Bytes transmitted on port
Meters added in the Pike release					
port	Gauge	port	port ID	Pollster	Existence of port
port.uptime	Gauge	s	port ID	Pollster	Uptime of port
port.receive.packets	Cumulative	packet	port ID	Pollster	Packets trasmitted on port
port.transmit.packets	Cumulative	packet	port ID	Pollster	Packets transmitted on port
port.receive.bytes	Cumulative	B	port ID	Pollster	Bytes received on port
port.transmit.bytes	Cumulative	B	port ID	Pollster	Bytes transmitted on port
port.receive.drops	Cumulative	packet	port ID	Pollster	Drops received on port
port.receive.errors	Cumulative	packet	port ID	Pollster	Errors received on port
switch.ports	Gauge	ports	switch ID	Pollster	Number of portson switch
switch.port.uptime	Gauge	s	switch ID	Pollster	Uptime of switch

These meters are available for OpenFlow based switches. In order to enable these meters, each driver needs to be properly configured.

VPN-as-a-Service (VPNaaS)

The following meters are collected for VPNaaS:

Name	Type	Unit	Resource	Origin	Note
Meters added in the Mitaka release or earlier					
network.services.vpn	Gauge	vpnservice	vpn ID	Pollster	Existence of a VPN
network.services.vpn.connec	Gauge	ipsec_site_conne	connection ID	Pollster	Existence of an IPSec connection

Firewall-as-a-Service (FWaaS)

The following meters are collected for FWaaS:

Name	Type	Unit	Resource	Origin	Note
Meters added in the Mitaka release or earlier					
network.services.firewall	Gauge	firewall	firewall ID	Pollster	Existence of a firewall
network.services.firewall.policy	Gauge	firewall_policy	firewall ID	Pollster	Existence of a firewall policy

Events

In addition to meters, the Telemetry service collects events triggered within an OpenStack environment. This section provides a brief summary of the events format in the Telemetry service.

While a sample represents a single, numeric datapoint within a time-series, an event is a broader concept that represents the state of a resource at a point in time. The state may be described using various data types including non-numeric data such as an instance's flavor. In general, events represent any action made in the OpenStack system.

Event configuration

By default, ceilometer builds event data from the messages it receives from other OpenStack services.

Note

In releases older than Ocata, it is advisable to set `disable_non_metric_meters` to `True` when enabling events in the Telemetry service. The Telemetry service historically represented events as metering data, which may create duplication of data if both events and non-metric meters are enabled.

Event structure

Events captured by the Telemetry service are represented by five key attributes:

event_type
A dotted string defining what event occurred such as `"compute.instance.resize.start"`.

message_id
A UUID for the event.

generated
A timestamp of when the event occurred in the system.

traits
A flat mapping of key-value pairs which describe the event. The event's traits contain most of the details of the event. Traits are typed, and can be strings, integers, floats, or datetimes.

raw
Mainly for auditing purpose, the full event message can be stored (unindexed) for future evaluation.

Event indexing

The general philosophy of notifications in OpenStack is to emit any and all data someone might need, and let the consumer filter out what they are not interested in. In order to make processing simpler and more efficient, the notifications are stored and processed within Ceilometer as events. The notification payload, which can be an arbitrarily complex JSON data structure, is converted to a flat set of key-value pairs. This conversion is specified by a config file.

> **Note**
>
> The event format is meant for efficient processing and querying. Storage of complete notifications for auditing purposes can be enabled by configuring `store_raw` option.

Event conversion

The conversion from notifications to events is driven by a configuration file defined by the `definitions_cfg_file` in the `ceilometer.conf` configuration file.

This includes descriptions of how to map fields in the notification body to Traits, and optional plug-ins for doing any programmatic translations (splitting a string, forcing case).

The mapping of notifications to events is defined per event_type, which can be wildcarded. Traits are added to events if the corresponding fields in the notification exist and are non-null.

> **Note**
>
> The default definition file included with the Telemetry service contains a list of known notifications and useful traits. The mappings provided can be modified to include more or less data according to user requirements.

If the definitions file is not present, a warning will be logged, but an empty set of definitions will be assumed. By default, any notifications that do not have a corresponding event definition in the definitions file will be converted to events with a set of minimal traits. This can be changed by setting the option `drop_unmatched_notifications` in the `ceilometer.conf` file. If this is set to `True`, any unmapped notifications will be dropped.

The basic set of traits (all are TEXT type) that will be added to all events if the notification has the relevant data are: service (notification's publisher), tenant_id, and request_id. These do not have to be specified in the event definition, they are automatically added, but their definitions can be overridden for a given event_type.

Event definitions format

The event definitions file is in YAML format. It consists of a list of event definitions, which are mappings. Order is significant, the list of definitions is scanned in reverse order to find a definition which matches the notification's event_type. That definition will be used to generate the event. The reverse ordering is done because it is common to want to have a more general wildcarded definition (such as `compute.instance.*`) with a set of traits common to all of those events, with a few more specific event definitions afterwards that have all of the above traits, plus a few more.

Each event definition is a mapping with two keys:

event_type
: This is a list (or a string, which will be taken as a 1 element list) of event_types this definition will handle. These can be wildcarded with unix shell glob syntax. An exclusion listing (starting with a `!`) will exclude any types listed from matching. If only exclusions are listed, the definition will match anything not matching the exclusions.

traits
: This is a mapping, the keys are the trait names, and the values are trait definitions.

Each trait definition is a mapping with the following keys:

fields
: A path specification for the field(s) in the notification you wish to extract for this trait. Specifications can be written to match multiple possible fields. By default the value will be the first such field. The paths can be specified with a dot syntax (`payload.host`). Square bracket syntax (`payload[host]`) is also supported. In either case, if the key for the field you are looking for contains special characters, like `.`, it will need to be quoted (with double or single quotes): `` payload.image_meta.`org.openstack__1__architecture` ``. The syntax used for the field specification is a variant of JSONPath

type
: (Optional) The data type for this trait. Valid options are: `text`, `int`, `float`, and `datetime`. Defaults to `text` if not specified.

plugin
: (Optional) Used to execute simple programmatic conversions on the value in a notification field.

Event delivery to external sinks

You can configure the Telemetry service to deliver the events into external sinks. These sinks are configurable in the `/etc/ceilometer/event_pipeline.yaml` file.

1.3.4 Management

Troubleshoot Telemetry

Logging in Telemetry

The Telemetry service has similar log settings as the other OpenStack services. Multiple options are available to change the target of logging, the format of the log entries and the log levels.

The log settings can be changed in `ceilometer.conf`. The list of configuration options are listed in the logging configuration options table in the Telemetry section in the OpenStack Configuration Reference.

By default `stderr` is used as standard output for the log messages. It can be changed to either a log file or syslog. The `debug` and `verbose` options are also set to false in the default settings, the default log levels of the corresponding modules can be found in the table referred above.

1.4 Ceilometer Configuration Options

1.4.1 Ceilometer Sample Configuration File

Configure Ceilometer by editing /etc/ceilometer/ceilometer.conf.

No config file is provided with the source code, it will be created during the installation. In case where no configuration file was installed, one can be easily created by running:

```
oslo-config-generator \
    --config-file=/etc/ceilometer/ceilometer-config-generator.conf \
    --output-file=/etc/ceilometer/ceilometer.conf
```

1.5 Ceilometer CLI Documentation

In this section you will find information on Ceilometers command line interface.

1.5.1 ceilometer-status

CLI interface for Ceilometer status commands

Synopsis

```
ceilometer-status <category> <command> [<args>]
```

Description

ceilometer-status is a tool that provides routines for checking the status of a Ceilometer deployment.

Options

The standard pattern for executing a **ceilometer-status** command is:

```
ceilometer-status <category> <command> [<args>]
```

Run without arguments to see a list of available command categories:

```
ceilometer-status
```

Categories are:

- `upgrade`

Detailed descriptions are below:

You can also run with a category argument such as `upgrade` to see a list of all commands in that category:

```
ceilometer-status upgrade
```

These sections describe the available categories and arguments for **ceilometer-status**.

Upgrade

ceilometer-status upgrade check

Performs a release-specific readiness check before restarting services with new code. For example, missing or changed configuration options, incompatible object states, or other conditions that could lead to failures while upgrading.

Return Codes

Return code	Description
0	All upgrade readiness checks passed successfully and there is nothing to do.
1	At least one check encountered an issue and requires further investigation. This is considered a warning but the upgrade may be OK.
2	There was an upgrade status check failure that needs to be investigated. This should be considered something that stops an upgrade.
255	An unexpected error occurred.

History of Checks

12.0.0 (Stein)

- Sample check to be filled in with checks as they are added in Stein.

CHAPTER

TWO

APPENDIX

2.1 Release Notes

2.1.1 Folsom

This is the first release (Version 0.1) of Ceilometer. Please take all appropriate caution in using it, as it is a technology preview at this time.

Version of OpenStack
It is currently tested to work with OpenStack 2012.2 Folsom. Due to its use of openstack-common, and the modification that were made in term of notification to many other components (glance, cinder, quantum), it will not easily work with any prior version of OpenStack.

Components
Currently covered components are: Nova, Nova-network, Glance, Cinder and Quantum. Notably, there is no support yet for Swift and it was decided not to support nova-volume in favor of Cinder. A detailed list of meters covered per component can be found at in *Measurements*.

Nova with libvirt only
Most of the Nova meters will only work with libvirt fronted hypervisors at the moment, and our test coverage was mostly done on KVM. Contributors are welcome to implement other virtualization backends' meters.

Quantum delete events
Quantum delete notifications do not include the same metadata as the other messages, so we ignore them for now. This isn't ideal, since it may mean we miss charging for some amount of time, but it is better than throwing away the existing metadata for a resource when it is deleted.

Database backend
The only tested and complete database backend is currently MongoDB, the SQLAlchemy one is still work in progress.

Installation
The current best source of information on how to deploy this project is found as the devstack implementation but feel free to come to #openstack-metering on OFTC for more info.

Volume of data
Please note that metering can generate lots of data very quickly. Have a look at the following spreadsheet to evaluate what you will end up with.

https://wiki.openstack.org/wiki/EfficientMetering#Volume_of_data

- *Folsom*

- Havana
- Icehouse
- Juno
- Kilo
- Liberty

Since Mitaka development cycle, we start to host release notes on Ceilometer Release Notes

2.2 Glossary

agent
Software service running on the OpenStack infrastructure measuring usage and sending the results to any number of target using the *publisher*.

billing
Billing is the process to assemble bill line items into a single per customer bill, emitting the bill to start the payment collection.

bus listener agent
Bus listener agent which takes events generated on the Oslo notification bus and transforms them into Ceilometer samples. This is the preferred method of data collection.

polling agent
Software service running either on a central management node within the OpenStack infrastructure or compute node measuring usage and sending the results to a queue.

notification agent
The different OpenStack services emit several notifications about the various types of events. The notification agent consumes them from respective queues and filters them by the event_type.

data store
Storage system for recording data collected by ceilometer.

meter
The measurements tracked for a resource. For example, an instance has a number of meters, such as duration of instance, CPU time used, number of disk io requests, etc. Three types of meters are defined in ceilometer:

- Cumulative: Increasing over time (e.g. disk I/O)
- Gauge: Discrete items (e.g. floating IPs, image uploads) and fluctuating values (e.g. number of Swift objects)
- Delta: Incremental change to a counter over time (e.g. bandwidth delta)

metering
Metering is the process of collecting information about what, who, when and how much regarding anything that can be billed. The result of this is a collection of "tickets" (a.k.a. samples) which are ready to be processed in any way you want.

notification
A message sent via an external OpenStack system (e.g Nova, Glance, etc) using the Oslo notifica-

tion mechanism[1]. These notifications are usually sent to and received by Ceilometer through the notifier RPC driver.

non-repudiable
"Non-repudiation refers to a state of affairs where the purported maker of a statement will not be able to successfully challenge the validity of the statement or contract. The term is often seen in a legal setting wherein the authenticity of a signature is being challenged. In such an instance, the authenticity is being "repudiated"." (Wikipedia,[2])

project
The OpenStack tenant or project.

polling agents
The polling agent is collecting measurements by polling some API or other tool at a regular interval.

publisher
The publisher is publishing samples to a specific target.

push agents
The push agent is the only solution to fetch data within projects, which do not expose the required data in a remotely usable way. This is not the preferred method as it makes deployment a bit more complex having to add a component to each of the nodes that need to be monitored.

rating
Rating is the process of analysing a series of tickets, according to business rules defined by marketing, in order to transform them into bill line items with a currency value.

resource
The OpenStack entity being metered (e.g. instance, volume, image, etc).

sample
Data sample for a particular meter.

source
The origin of metering data. This field is set to "openstack" by default. It can be configured to a different value using the sample_source field in the ceilometer.conf file.

user
An OpenStack user.

[1] http://en.wikipedia.org/wiki/Ceilometer
[2] https://opendev.org/openstack/oslo.messaging/src/branch/master/oslo_messaging/notify/notifier.py